

Noetherian Induction for Computer-Assisted First-Order Reasoning

Sorin Stratulat

▶ **To cite this version:**

Sorin Stratulat. Noetherian Induction for Computer-Assisted First-Order Reasoning. Symbolic Computation [cs.SC]. Université de Lorraine, 2021. tel-03286314

HAL Id: tel-03286314
https://hal.science/tel-03286314

Submitted on 14 Jul 2021

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

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

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Noetherian Induction for Computer-Assisted First-Order Reasoning

Mémoire d'habilitation à diriger des recherches

defended on 29^{th} of June 2021

Université de Lorraine

(Speciality of Computer Science)

by

Sorin STRATULAT

Committee

President :	*Olga Kouchnarenko, Professor, Université de Franche-Comté, France*
Reviewers :	Adel Bouhoula, Professor, Arabian Gulf University, Bahreïn Evelyne Contejean, Researcher, CNRS, France Viorica Sofronie-Stokkermans, Professor, Koblenz University, Germany
Examiners :	Tudor Jebelean, Professor, Johannes Kepler University, Austria Olga Kouchnarenko, Professor, Université de Franche-Comté, France Stephan Merz, Researcher, INRIA, France Michaël Rusinowitch, Researcher, INRIA, France Jeanine Souquières, Professor, Université de Lorraine, France

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thesul.

Contents

General Introduction

1 Motivation vii
2 Some induction principles for first-order reasoning viii
3 Term-based Noetherian induction reasoning and its limitations x
4 Formula-based Noetherian induction reasoning and its advantages xi
5 Goals and structure of the content xiv
6 List of main contributions xv

Part I First-Order Noetherian Induction Proving xvii

Chapter 1
First-Order Instances of the Noetherian Induction Principle 1

1.1 Basic notions 1
1.2 The term- and formula-based instances 3
1.3 Term-based Noetherian induction principles 5
1.4 Formula-based Noetherian induction principles 6
1.5 Conclusions 10

Chapter 2
A Cyclic and Non-reductive Formula-based Noetherian Induction Proof Method

2.1 The inference system 13
2.2 The DRaCuLa proof strategy 14
2.3 Building cycles 17
2.4 Managing mutual induction 21
2.5 Representing implicit induction proofs as cyclic proofs 22
2.6 Conclusions 24

Chapter 3
Building Explicit Induction Proofs for Conjectures Proved with Cyclic Induction

3.1 Coq experiments 26
3.1.1 Proving by cyclic induction 27
3.1.2 Proving by structural induction 29

3.2 Lazy generation of induction schemas for algorithm synthesis . . . 36
3.2.1 Sorting of binary trees . . . 37
3.2.2 Synthesis problem and method . . . 38
3.2.3 Induction principles and algorithm extraction . . . 38
3.2.4 Refining induction by lazy reasoning . . . 39
3.3 Conclusions . . . 41

Part II Mechanising and Certifying Noetherian Induction Reasoning 43

Chapter 4
Tool: the SPIKE Prover

4.1 The inference system and proof strategies . . . 46
4.1.1 The 'reasoning by implicit induction' setting . . . 46
4.1.2 The inference system . . . 46
4.1.3 The layout of a specification file . . . 49
4.2 A complete example of SPIKE specification . . . 50
4.3 Conclusions . . . 52

Chapter 5
Application: Validation of the JavaCard Platform

5.1 A primer on CertiCartes . . . 54
5.2 Improvements of SPIKE . . . 55
5.2.1 At the specification language . . . 55
5.2.2 At the proof engine . . . 56
5.3 Applications to JavaCard . . . 58
5.3.1 CDO . . . 58
5.3.2 CDA and MON . . . 58
5.3.3 Assessment . . . 58
5.4 Conclusions and related works . . . 61

Chapter 6
Certifying Formula-based Noetherian Induction Reasoning

6.1 Formalising formula-based Noetherian induction proofs . . . 64
6.1.1 Formalising the induction ordering and measure values with COCCINELLE . 64
6.1.2 Proving formulas from LF . . . 66
6.2 Certifying implicit induction proofs . . . 67
6.2.1 Inference systems for implicit induction . . . 67
6.2.2 A first attempt to certify the proof of $q(x, y) = true$ . . . 68
6.2.3 Coq formalizations of implicit induction proofs . . . 69
6.2.4 Certification of implicit induction proofs for Coq formalisations using the functional programming style . . . 72

6.3 Certifying cyclic proofs . . . 74
6.3.1 Inference systems for cyclic induction . . . 74
6.3.2 Cyclic proofs . . . 76
6.3.3 Certification of cyclic proofs for Coq formalisations using the logic programming style . . . 79
6.3.4 Certification of cyclic proofs for Coq formalisations using the functional programming style . . . 80
6.4 Automatic certification of SPIKE proofs . . . 84
6.5 Conclusions . . . 84

Part III Bridges with Other Reasoning Techniques 87

Chapter 7
Cyclic Induction Reasoning for FOL with Inductive Definitions

7.1 The logical framework . . . 90
7.1.1 Checking the soundness of pre-proofs . . . 92
7.2 Defining the ordering-based checking criteria . . . 93
7.2.1 Normalising pre-proof trees . . . 93
7.2.2 Building the digraph of a pre-proof tree-set . . . 95
7.2.3 Defining the ordering and derivability conditions . . . 97
7.3 Implementation . . . 101
7.4 Converting cyclic to Noetherian induction reasoning . . . 102
7.4.1 The Cyclist proof . . . 102
7.4.2 The conversion procedure . . . 103
7.4.3 Experimental results . . . 105
7.5 Conclusions . . . 106

Chapter 8
Saturation-based Reasoning

8.1 The logical framework . . . 108
8.1.1 Contextual cover sets (CCSs) . . . 108
8.1.2 The A^1 and A^2 inference systems . . . 109
8.1.3 Reasoning modules . . . 110
8.1.4 Methodology for designing and analysing CCS-based inference systems . . . 110
8.1.5 Case study: designing an implicit induction inference system . . . 111
8.2 Analysing and extending saturation-based inference systems . . . 113
8.2.1 Case study 1: a paramodulation-based inference system . . . 113
8.2.2 Case study 2: a resolution-based inference system . . . 115
8.3 Conclusions . . . 119

Future Works

C.1 Detailed project: certification of cyclic reasoning by converting cyclic to explicit induction proofs . . . 123
C.1.1 The LKID and LKID$_a$ explicit induction inference systems . . . 123
C.1.2 Overview of the procedure . . . 124
C.1.3 Generating the explicit induction schema . . . 125
C.1.4 Generating the explicit induction proof . . . 127
C.1.5 Checking the admissibility property . . . 129
C.1.6 Certifying other cyclic proofs . . . 134
C.1.7 Further lines of research . . . 135
C.2 Other projects . . . 137
C.2.1 Strategies for directly building sound CLKID$^{\omega}_{N}$ pre-proofs . . . 137
C.2.2 Certification of saturation-based proofs . . . 139
C.2.3 Applications . . . 139

Bibliography **141**

Index **151**

Glossary **153**

List of Figures

1 The cases when Hercules wins. . . . viii
2 The cases when Hercules cuts the necks of Hydra. . . . ix
3 The digraph representation of the Coq pre-proof from Example 2. . . . xiii
4 The digraph resulting after the duplication of the node labelled by (**P** x). . . . xiii

2.1 A cycle for checking n IHs. . . . 16
2.2 The skeleton of the D_c-proof. . . . 22

3.1 The digraph of the cyclic proof of $\forall x\ u\ v, R(x,u,v)$, for any $R \in \mathcal{R}$. . . . 27

5.1 Commutative diagram of defensive and offensive execution. . . . 55
5.2 Commutative diagram of defensive and abstract execution. . . . 55
5.3 New inference rules. . . . 57
5.4 An excerpt of a SPIKE specification formalizing the instruction CONV. . . . 59

6.1 The I_c^b-preproof of $\{q(x,y) = true\}$ as an oriented graph. . . . 76
6.2 The transformation of a non-root IH-node. . . . 77
6.3 The normalised I_c^b-preproof of $\{q(x,y) = true\}$. . . . 77
6.4 The graph representation of the I_c^f-preproof of $\{e'_{13}(u_1,u_2,u_3)\}$. . . . 82

7.1 Sequent-based rules for classical first-order logic. . . . 91
7.2 Sequent-based rules for equality reasoning. . . . 91
7.3 The result of the first operation. . . . 94
7.4 The result of the second operation. . . . 94
7.5 A case when the comparison tests fail while the model checker succeeds. . . . 103
7.6 The screenshot of the CYCLIST pre-proof built for the P&Q conjecture. . . . 104

8.1 The one-step inference rules. . . . 109
8.2 The two-step inference rules. . . . 110
8.3 The inference system P. . . . 112
8.4 The EXPANDREWRITE rule. . . . 113
8.5 The inference system G. . . . 114
8.6 The one-step A_s-inference rules. . . . 115
8.7 RP: the original inference system with triplets as proof states. . . . 118
8.8 RP': the inference system RP with proof states under the form *(conjectures, premises)*. . . . 119
C.9 The input pre-proof tree-set. . . . 124
C.10 The main steps of the LKID$_a$ proof of $S(N_r) \equiv \Gamma \vdash \Delta$. . . . 127
C.11 The beginning part of the LKID$_a$ proof of $\vdash Pu \wedge Qxy$. The terminal nodes are individual ICs. . . . 129
C.12 The pre-proof tree-set for showing the admissibility of $P_\pi uxy$. . . . 132

General Introduction

1 Motivation

(Mathematical) induction is a most successful proof technique to reason on data structures of unbounded size like naturals and lists. Already known by the ancient greeks and rediscovered by Pascal and Fermat at the second half of the seventeenth century, it became a precious formal tool for mathematicians to such an extent that Poincaré considered it, at the beginning of the twentieth century, as the mathematical reasoning *par excellence* [Poincaré, 1902]. This is also the case for computer science. The fundamental notion of computable function can be defined in many equivalent models of computation, among which the Turing machines [Turing, 1936] and the (μ-)recursive functions [Kleene, 1936]. There is a strong link between recursion and induction; for example, the correctness of primitive recursive functions is proved by induction on naturals.

After the inception of computers, McCarthy saw an interest for its mechanization and coded into Lisp the recursion induction [McCarthy, 1963], a method for proving the equivalence between two recursively defined functions. Later on, Burstall [Burstall, 1969] showed the importance of structural schemata-based induction to verify properties about recursively defined data structures. Since then, a lot of 'proof by induction' methods have been proposed and contributed to many successful stories about the validation and verification of (critical) programs. A closer look on the applications of induction-based theorem provers as ACL2 [Kaufmann and Moore, 1999][1] may identify the following (non-exhaustive list of) application domains:

- data structures (fully ordered sets, finite sets, powerlists),
- processor modelling and hardware verification (assynchronous circuits, self-timed circuits, x86 instruction set architecture, flash memories, AMD processors, pipelined machines),
- programming languages and software verification (Java-like bytecode, cryptographic language compilers, verification condition generation, program verification strategies),
- floating point and real arithmetic (register-transfer logic, continuity and differentiability, floating point operations: addition, division, square root),
- concurrency (cache coherence, interactive consistency),
- model checking (predicate abstraction, reduction of invariant proofs, compositional and μ- calculus model checking), and
- logic and metamathematics (quadratic unification, Higman's lemma, Dickson's lemma, synthesized SAT-provers, decision procedures for propositional logic, rewriting, computer algebra).

Some repositories[2] contain benchmarks and challenge problems for induction-based theorem provers. An example of non-trivial property that can be proved by induction is $rev(rev(l)) = l$, for every list l, where rev is the function that computes the reverse of a list.

[1] https://www.cs.utexas.edu/users/moore/publications/acl2-papers.html
[2] e.g., https://tip-org.github.io/

We are interested in the use of induction to reason on programs specified in a first-order language.

2 Some induction principles for first-order reasoning

Peano induction is one of the classical examples of induction principles: to prove a formula $P(x)$, for any natural x, it is enough to prove both $P(0)$ and $P(S(x'))$, where x' is a fresh natural variable, by allowing to use $P(x')$ in the proof of $P(S(x'))$. The power of induction reasoning stands in the use of 'not-yet proved' facts, as $P(x')$.

Other induction principles, adapted for first-order reasoning, are issued from the Knuth-Bendix saturation-based completion procedure [Knuth and Bendix, 1970]. It opened the way to rewrite-based automated reasoning methods like the inductionless induction [Musser, 1980, Goguen, 1980, Huet and Hullot, 1980, Lankford, 1980] and other reductive induction methods as implicit induction [Kounalis and Rusinowitch, 1990a, Reddy, 1990]. A more recent induction reasoning technique for first-order logic with inductive definitions (FOL_{ID}) is cyclic induction [Brotherston and Simpson, 2011].

Not all induction reasoning techniques are equivalent when proving properties. In the frame of FOL_{ID}, it has been shown in [Berardi and Tatsuta, 2019] that Peano-like induction reasoning is not as powerful as cyclic induction reasoning, by using arguments based on the 2-Hydra problem.

The 2-Hydra problem is a particular case showing the termination of the battle between Hercules and Hydra [Dershowitz and Moser, 2007] when Hydra has at most two heads that hang on the top of necks of different lengths. Hercules prevails if either Hydra has i) no heads at all, or ii) the length of the first neck is 1 unit and it has lost the second head (i.e., the length of the neck is 0), or iii) the length of the second neck is 1 unit, as in Figure 1.

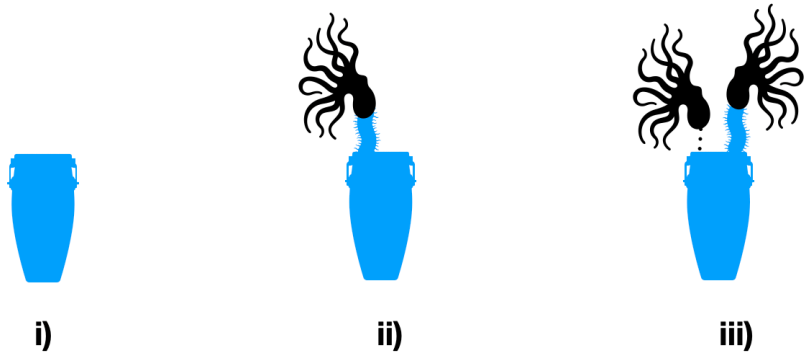


Figure 1: The cases when Hercules wins.

Hercules can cut the Hydra's necks according to the following rules. If both necks have strictly positive lengths, then Hercules can cut them such that the first neck is shorter by 1 unit and the second by 2 units (see the case iv in Figure 2). If Hydra has already lost one of the heads and the neck of the other head has a length l of at least 2 units, the first head will have a neck of length $l-1$ units and the second head a neck of length $l-2$ units (see the cases v and vi in Figure 2).

In FOL_{ID}, the six cases can be formalised as the axioms of a binary inductive predicate p:

i) $p(zero, zero)$,

ii) $p(succ(zero), zero)$,

iii) $\forall x,\ p(x, succ(zero))$,

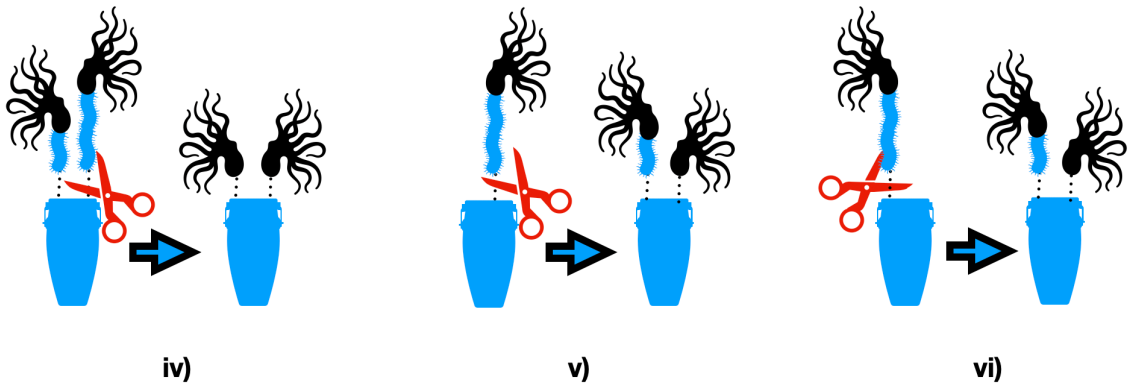


Figure 2: The cases when Hercules cuts the necks of Hydra.

iv) $\forall x\ y,\ p(x,y) \to p(succ(x), succ(succ(y)))$,

v) $\forall y,\ p(succ(y), y) \to p(zero, succ(succ(y)))$, and

vi) $\forall x,\ p(succ(x), x) \to p(succ(succ(x)), zero)$,

where *zero* and *succ* (the 'successor' function) are intended to represent the usual constructors of natural numbers. In this setting, Hercules always wins over Hydra if $\forall\ x\ y,\ p(x,y)$ holds whenever x and y are natural numbers, where the property of being a natural number is captured by another inductive predicate. In [Berardi and Tatsuta, 2019], it has been shown that this property cannot be proved by induction principles similar to Peano induction.

The Noetherian induction principle. Peano induction is a particular case of a more general induction principle called *Noetherian induction* or well-founded induction. It allows to prove the validity of a property ϕ for any element from a potentially infinite poset $(\mathcal{E}, <)$, provided that $<$ is a *well-founded* ordering which excludes the occurrence of any infinite strictly decreasing sequence of elements. The Noetherian induction principle can be formally stated as follows:

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

Thanks to the well-founded property of the ordering, the assumptions $\phi(k)$, called *induction hypotheses* (IHs), can be soundly applied in the proof of the *induction conclusion* $\phi(m)$. The soundness proof will be shown for the contrapositive version of the Noetherian induction, called the 'Descente Infinie' (or Infinite Descent) induction, but a similar proof can be built also for the soundness of the Noetherian induction principle.

'Descente Infinie' induction. $(\forall m \in \mathcal{E}, \neg\phi(m) \Rightarrow (\exists k \in \mathcal{E}, k < m \wedge \neg\phi(k))) \Rightarrow \forall p \in \mathcal{E}, \phi(p)$.

The soundness proof is by contradiction. If there exists an element $m_0 \in \mathcal{E}$ such that $\neg\phi(m_0)$, according to the induction principle, there is a smaller element m_1 such that $\neg\phi(m_1)$, for which there is an even smaller element m_2 such that $\neg\phi(m_2)$, and so on. In this way, an infinite strictly descending sequence of elements of $\mathcal{E}$ is built. This contradicts the well-foundedness property of the ordering.

In a first-order setting, two useful classes of Noetherian/'Descente Infinie' induction instances are distinguished, for which the elements of $\mathcal{E}$ are i) (vectors of) terms, and ii) (first-order) formulas. For the latter case, ϕ can be the second-order identity predicate in order to keep up with the first-order setting.

3 Term-based Noetherian induction reasoning and its limitations

Term-based instances of the Noetherian induction principle include the conventional induction methods, based on induction schemas that explicitly define the IHs to be used in the proof of each induction conclusion. The information justifying the soundness property is *locally* embedded inside the induction schemas, hence their natural integration into deductive, sequent-based *inference systems* in terms of *induction rules*. During a proof, it may happen to define useless IHs or to ask for crucial IHs that are not yet defined. The case of mutual induction, for which instances of a formula are used as IHs in the proof of other formulas, and viceversa, is hardly manageable with non-mutual recursion and leads to more technical and complex proofs.

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.

Today, it is hardly imaginable a modern theorem prover not integrating 'proof by induction' features. Explicit induction schemas can be automatically generated from the analysis of inductive predicate, recursive function or datatype definitions, a feature that is implemented by many formal reasoning tools like the Coq proof assistant [The Coq development team, 2020]. The following example outlines, by the means of Coq, the main problems that may arise when using Peano-like induction principles, and explicit induction reasoning in general, for proving properties about mutually defined functions.

Example 1 (Coq specification and proofs of the P&Q example) *This example has been adapted from [Wirth, 2004]. By using the specification language of Coq, let us define two mutually defined inductive predicates* **P** *and* **Q**, *as well as* **N** *intended to define natural numbers. Let us also provide the main steps of the Coq proof script of the lemma* p_is_true, *stating that* **P** *holds for any natural, to be used later to prove our main theorem stating that* (**Q** x y) *holds for every natural* x *and* y:

```
Variable T: Set.
Variable zero: T.
Variable succ: T → T.

Inductive P: T → Prop :=
  p1: P zero
| p2: ∀ x, (P x ∧ Q x (succ x)) → P (succ x)
with
  Q: T → T → Prop :=
  q1: ∀ y, Q y zero
| q2: ∀ x y, (Q x y ∧ P x) → Q x (succ y).

Inductive N: T → Prop :=
  n1: N zero
| n2: ∀ x, N x → N (succ x).
```

```
Theorem p_is_true : ∀ u, N u → P u.
Proof.
  [...].
  induction H. (* u is the induction variable *)
  - Case "zero". apply p1.
  - Case "succ". apply p2. [...].
    generalize (succ x).
    [...].
    induction H1. (* second induction step *)
    + SCase "zero". apply q1.
    + SCase "succ". apply q2. [...].
Qed.
```

The type T *is generic and the variables 'zero' and 'succ' are meant again to be the constructors of naturals. Each inductive definition is built from labelled axioms that represent either atoms or implication formulas.*

The Coq proof scripts are not meant to be read *but* executed *using Coq proof environments as CoqIDE.*[3] *The advantage of using proof environments is due to a better undertanding of each proof step by displaying the current proof state. In the following, we will explain to the readers of* p_is_true*'s proof*

[3] https://coq.inria.fr/refman/practical-tools/coqide.html

script[4] *the main induction proof steps. The induction principle, denoted by* N_ind*, is called using the* `induction` *command. The proof scripts of the resulting induction cases are indented and prefixed by a symbol, e.g., '-' or '+', followed by some comment, e.g., Case "zero", related the current case. Coq builds automatically* N_ind *from the recursive definition of* **N***:*

```
forall P : T -> Prop,
P zero ->
(forall x : T, N x -> P x -> P (succ x)) ->
forall t : T, N t -> P t
```

which has the same induction cases as the Peano induction principle.

The induction cases are built depending on the argument given to `induction` *which is a label of a premise formula of the form* (**N** v)*, where* v *is the induction variable to be instantiated by the induction schema. If* P *is a property defined over the terms* t *satisfying* (**N** t)*,* N_ind *states that* $(P\ t)$ *holds for all terms* t *of type* T *if both induction cases* $(P\ zero)$ *and* $(P\ (succ\ x))$ *hold, where* x *is a fresh variable. It also states that* $(P\ x)$ *can be soundly used in the proof of* $(P\ (succ\ x))$ *as IH. It can be noticed that the proof requires two induction steps, one embedded into the other. The first step uses* u *as induction variable, while the second step uses as induction variable the generalization of the term* $(succ\ x)$*. The notation* [...] *means that some non-relevant Coq script is missing or some IH was applied at (S)Case "succ".*

The lemma can be used in the proof of our main theorem. The proof requires only one induction step that applies one more time the N_ind *induction principle:*

```
Theorem q_is_true : ∀ u v, N u → N v → Q u v.
Proof.
  [...]. induction H0. (* v is the induction variable *)
  - Case "zero". apply q1.
  - Case "succ". apply q2.
    [...]. apply p_is_true. [...].
Qed.
```

It is hard to find a global proof for the main theorem from Example 1 where the *N_ind* induction principle is applied only once. This is because the proof of $\forall u, (\mathbf{P}\ u)$ needs an instance of $\forall x\ y, (\mathbf{Q}\ x\ y)$, and vice versa. In another proof attempt, one can take advantage of the fact that the main theorem is a consequence of the conjunction $\forall x\ y, (\mathbf{P}\ x) \wedge (\mathbf{Q}\ x\ y)$. However, the proof of the conjunction lemma is as difficult as the global proof, requiring also three induction steps. It can be noticed that the proof attempts for other conjunction lemmas, as $\forall x\ y, (\mathbf{P}\ y) \wedge (\mathbf{Q}\ x\ y)$ for which the shared variable is y, and $\forall u\ x\ y, (\mathbf{P}\ u) \wedge (\mathbf{Q}\ x\ y)$, where no variable is shared, will fail.[5]

From the previous proof attempts, we conclude that one of the main difficulties encountered when using the term-based approach is to devise successful induction schemas.

4 Formula-based Noetherian induction reasoning and its advantages

Simpler proof derivations, based only on variable instantiations and unrolls of the definitions of **P** and **Q**, can be built using formula-based instances of the Noetherian induction principle. This approach uses lazy and mutual induction reasoning to stop the proof development of formulas if they are instances of

[4]The full Coq script is available at https://drive.google.com/file/d/10HUmH4ywl1O1PSPjODH58zXwL6K_KBeL/view?usp=sharing

[5]The details of the Coq proof can also be accessed at https://drive.google.com/file/d/10HUmH4ywl1O1PSPjODH58zXwL6K_KBeL/view?usp=sharing

previously generated formulas. The result of such a terminating proof development is referred to as a *pre-proof*.

Example 2 (a Coq pre-proof of the P&Q example)

```
Theorem cyclic: ∀ x y, N x → N y → Q x y.
Proof.

[...].
    (* instantiate y from (Q x y) *)
inversion H0. [...].
    (* case (Q x zero) *) apply q1. [...].
    (* case (Q x (succ y')) *) apply q2. [...].
    (* the proof of (Q x y') requires induction reasoning *)
Focus 2.
    (* instantiate x from (P x) *)
inversion H. [...].
    (* case (P zero) *) apply p1. [...].
    (* case (P (succ x')) *) apply p2. [...].
    (* the proof of (P x') requires induction reasoning *)
Focus 2.
    (* the proof of (Q x' (succ x')) requires induction reasoning *)
Admitted.
```

The `inversion` *tactic is a different way to instantiate variables from an inductive atom $P(\overline{a})$ given as a premise, also derived from the definition of the inductive predicate P. 'Focus 2' is used to process the next branch in the tree structure of the Coq pre-proof. The command 'Admitted' informs Coq to abandon the proof. As it is, Coq is not able to validate the above proof script.*

Checking the soundness of pre-proofs. Not every pre-proof is *sound*. In our context, the soundness property boils down to validate the usage of IHs by the application of some induction principle. A formula-based induction *proof* is a pre-proof that satisfies the ordering constraints required by the application of formula-based Noetherian induction principles. The validation of IHs may be tricky because one have to find the orderings that satisfy the ordering constraints. To show how the ordering constraints are built, we will represent the pre-proofs as *directed graphs* (digraphs).

Example 3 (a formula-based induction proof of the P&Q example) *Figure 3 is a digraph representation of the Coq pre-proof from Example 2, where the nodes represent atoms of the form ($\mathbf{P}$ t) and ($\mathbf{Q}$ t_1 t_2) found in the conclusion of the main steps of the pre-proof. Each downward arrow link a node with the new nodes issued after the process of the node.*

The pre-proof starts by instantiating y by $zero$ and $(succ\ y)$ from the root node labelled by ($\mathbf{Q}$ x y). ($\mathbf{Q}$ x $zero$) is true, by q1. The instantiating substitutions label the corresponding downward arrows. The instance ($\mathbf{Q}$ x $(succ\ y')$) is also true if ($\mathbf{P}$ y') and ($\mathbf{Q}$ x y') are true, by q2. No further transformation is supported by ($\mathbf{Q}$ x y') which is an instance of the root node using the substitution $\{x \mapsto y'\}$. The instantiation relation is graphically represented by an upward dashed arrow labelled by the instantiating substitution written in boldface style. The variable x of ($\mathbf{P}$ x) is further instantiated by $zero$ and $(succ\ x')$. ($\mathbf{P}$ $zero$) is true, by p1, while ($\mathbf{P}$ $(succ\ x')$) is true if both ($\mathbf{Q}$ x' $(succ\ x')$) and ($\mathbf{P}$ x') are true, by p2. ($\mathbf{P}$ x') is an instance of ($\mathbf{P}$ x') using the substitution $\{x \mapsto x'\}$, while ($\mathbf{Q}$ x' $(succ\ x')$) is an instance of the root node using the substitution $\{x \mapsto x'; y \mapsto (succ\ x')\}$.

Sections 6.2 and 6.3 will show different ways to apply the formula-based Noetherian induction principle in order to validate the IHs. We will follow the approach detailed in Section 6.3. Let the set $\mathcal{E}$, used in the definition of the formula-based Noetherian induction, consist of the two formulas $\forall t, (\mathbf{P}\ t)$ and $\forall\ t_1\ t_2, (\mathbf{Q}\ t_1\ t_2)$. This approach requires that the nodes labelled by ($\mathbf{P}$ x) and ($\mathbf{Q}$ x y) be root nodes. This is already the case for the node labelled by ($\mathbf{Q}$ x y) but not for ($\mathbf{P}$ x). The node labelled by ($\mathbf{P}$ x) can be

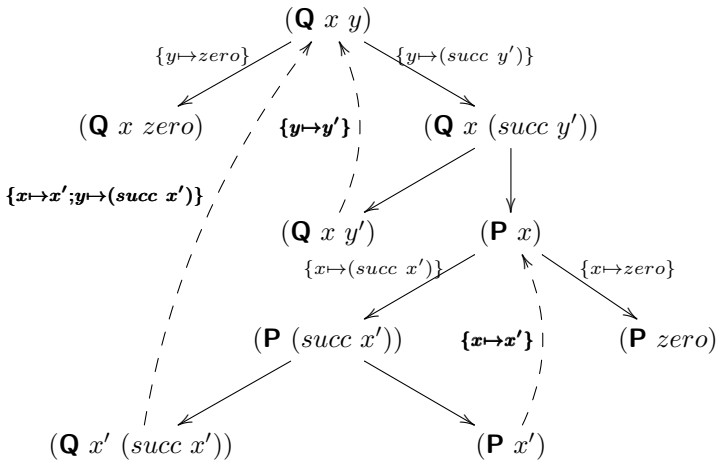


Figure 3: The digraph representation of the Coq pre-proof from Example 2.

duplicated such that one instance is a terminal node in the digraph and the subtree rooted by the other instance is detached from the digraph to become a new tree. An upward dashed arrow is added to point the first instance to the second one. The arrow is labelled by an identity substitution which instantiates x by itself. The final result is the digraph from Figure 4, consisting of two trees.

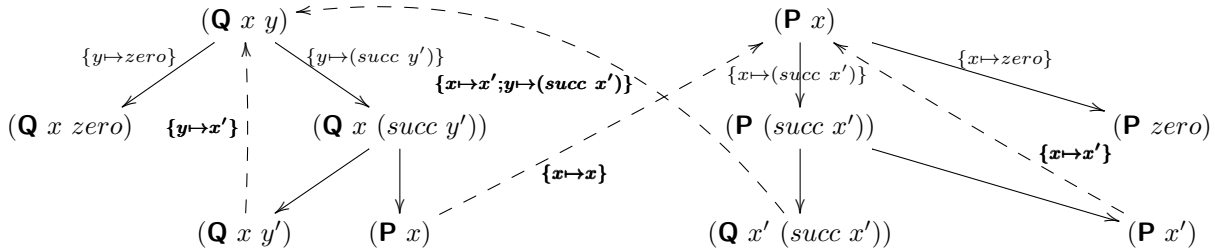


Figure 4: The digraph resulting after the duplication of the node labelled by (**P** x).

The new digraph is normalized such that all the IHs label terminal nodes that point to root nodes. We say that an IH labelling a terminal node n is validated *if it is smaller than some instance of the root formula of the tree to which n belongs. This instance uses the* cumulative substitution *resulting from the composition of the substitutions that label the downward arrows encountered along the path leading the root to n. Since the IHs are all instances of root formulas, the ordering constraints will compare only instances of root formulas. If $P\sigma$ is the instance of a formula P with the substitution σ, the ordering constraints for the four IHs of our example are:*

- *IH* $(\mathbf{Q}\ x\ y')$*:* $(\mathbf{Q}\ x\ y)\{x \mapsto x; y \mapsto y'\} <_p (\mathbf{Q}\ x\ y)\{x \mapsto x; y \mapsto (succ\ y')\}$,
- *IH* $(\mathbf{P}\ x')$*:* $(\mathbf{P}\ x)\{x \mapsto x'\} <_p (\mathbf{P}\ x)\{x \mapsto (succ\ x')\}$,
- *IH* $(\mathbf{Q}\ x'\ (succ\ x'))$*:* $(\mathbf{Q}\ x\ y)\{x \mapsto x'; y \mapsto (succ\ x')\} <_p (\mathbf{P}\ x)\{x \mapsto (succ\ x')\}$*, and*
- *IH* $(\mathbf{P}\ x)$*:* $(\mathbf{P}\ x)\{x \mapsto x\} <_p (\mathbf{Q}\ x\ y)\{x \mapsto x; y \mapsto (succ\ y')\}$.

The ordering $<_p$ has to be defined such that the ordering constraints, resulting after the application of the substitutions to the root formulas, are satisfied: $(\mathbf{Q}\ x\ y') <_p (\mathbf{Q}\ x\ (succ\ y'))$, $(\mathbf{P}\ x') <_p (\mathbf{P}\ (succ\ x'))$, $(\mathbf{Q}\ x'\ (succ\ x')) <_p (\mathbf{P}\ (succ\ x'))$ *and* $(\mathbf{P}\ x) <_p (\mathbf{Q}\ x\ (succ\ y'))$*. If the measure value of a formula*

($\boldsymbol{P}$ t) (resp., ($\boldsymbol{Q}$ t_1 t_2)) is the multiset $\{t, t\}$ (resp., $\{t_1, t_1, t_2\}$), the ordering constraints are boiled down to the following comparisons between multisets of terms: $\{x, x, y'\} <_p \{x, x, (succ\ y')\}$, $\{x', x'\} <_p \{(succ\ x'), (succ\ x')\}$, $\{x', x', (succ\ x')\} <_p \{(succ\ x'), (succ\ x')\}$, and $\{x, x\} <_p \{x, x, (succ\ y')\}$. It can be shown that the constraints are satisfied if $<_p$ is the multiset extension of a multiset path ordering (mpo) [Baader and Nipkow, 1998] based on the precedence over the function symbols stating that 'zero' is smaller than 'succ'.

Several formula-based instances of the Noetherian induction principles can be applied for checking whether a pre-proof is a proof. Validating the IHs can be done *after* building the pre-proofs or *during* their development. Also, the ordering constraints may be implicitly satisfied if *reductive* proof procedures are used. The attribute 'reductive' relates to the fact that *every* inference step transforms formula instances into strictly smaller ones or, sometimes, smaller or equal ones. An example of reductive induction technique is the rewrite-based *implicit induction*, suggested in [Kounalis and Rusinowitch, 1990b] and defined later in [Bronsard *et al.*, 1994]. In this case, there is only one application of the formula-based induction principle by using a unique induction ordering defined over all the formulas encountered during the proof derivation. 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 SPIKE, an implicit induction prover [The SPIKE development team, 2020].

The non-trivial induction reasoning requires the usage of IHs. Intrinsically, it is *cyclic* in the sense that it includes cases where the proof of a formula ϕ depends on (an instance of) ϕ given as IH. In some cases, the validity of IHs can also be checked without employing Noetherian induction reasoning and ordering constraints. An example is the CLKID$^\omega$ inference system [Brotherston and Simpson, 2011] that can reason on FOL_{ID}. Its implementation in the Cyclist prover [Brotherston *et al.*, 2012] uses a proof strategy guided by building cycles of formulas that satisfy some *global trace condition* and automata-based methods for checking its validity.

The manual check of the validity of IHs may be tedious, error-prone and costly. For example, the automatically generated implicit induction proofs may contain thousands of proof steps that make hard to explicitly state and verify the ordering constraints. For these reasons, their mechanical certification becomes a necessity. 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 the direct application of instances of the Noetherian induction principle. In Coq, Noetherian induction principles can also be derived from recursion-based specifications. They 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 inductive 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, but no termination proof is needed.

5 Goals and structure of the content

Goals. Our main objectives are: i) to understand the relations between term- and formula-based induction principles in order to bridge the gap between the underlying proof methods, ii) to free the inductive reasoning from computation, and iii) implement it more effectively and trustworthily.

Structure of the content and contributions. The rest of the document is structured in three parts. The first part is more theoretical and has three chapters. Chapters 1 and 2 are based on the work published in [Stratulat, 2012]. Chapter 1 gives an overview of the term- and formula-based instances of the Noetherian induction principle. Chapter 2 introduces a cyclic and non-reductive formula-based induction proof method and its relation with reductive methods as those based on implicit induction. The

last chapter, based on [Stratulat, 2016, Dramnesc *et al.*, 2019], studies the relations between term- and formula-based proofs. Firstly, we show how to implement cyclic reasoning in Coq and identify conjectures that are, on the one hand, easy to be proved by cyclic induction but are, on the other hand, hard to be proved by using explicit induction. The last part of Chapter 3 shows how cyclic reasoning was converted to explicit induction reasoning and mechanized during a concrete application: the synthesis of sorting algorithms for binary trees.

The second part also has three chapters but is more practical. Chapter 4 introduces the SPIKE prover, based on [Stratulat, 2020]. Chapter 5, synthesizing results from [Barthe and Stratulat, 2003], presents an industrial-size application of the implicit induction and SPIKE: the validation of the JavaCard platform. Chapter 6, based on [Stratulat, 2010, Stratulat and Demange, 2011, Henaien and Stratulat, 2013, Stratulat, 2017b], gives a methodology for the Coq certification of implicit and cyclic induction reasoning for recursive-based specifications using the functional and logic programming styles.

The last part, consisting of two chapters, reveals bridges of the Noetherian induction reasoning with other reasoning techniques. Chapter 7, based on [Stratulat, 2017a, Stratulat, 2018], introduces a different method to validate cyclic FOL_{ID} proofs, via ordering constraints. This approach allows to represent cyclic FOL_{ID} induction reasoning as Noetherian induction reasoning that can be validated by Coq. The second chapter presents a connection between saturation-based and Noetherian induction reasoning, published in [Stratulat, 2005, Stratulat, 2007].

The document ends with proposals for future projects.

6 List of main contributions

We briefly highlight the main contributions related to the use of the Noetherian induction principle in first-order reasoning.

Taxonomy for the first-order instances of the Noetherian induction principle. In Chapter 1, we proposed a classification of the first-order instances of the Noetherian induction principle into term- and formula-based instances. This allowed us to classify different induction reasoning techniques and induction principles for first-order reasoning. We showed that every term-based proof can be directly converted to a formula-based one. In the other direction, the direct conversion is possible only for a class of formula-based proofs. Section 3.1 gave several examples of formula-based proofs that are hard or impossible to be proved using term-based induction principles that may be issued from the formula-based proofs, e.g., using the same variable instantiation schemas. On the other hand, Section 3.2 presented a class of formula-based proofs that can be translated into term-based ones. The question whether every formula-based proof can be converted to a term-based one remains open. Our effort to better understand the relations between formula- and term-based instances is continued in the detailed project from Section C.1.

Formula-based induction proof technique. In Chapter 2, we proposed a cyclic and non-reductive formula-based reasoning technique and applied it to prove conjectures about conditional specifications. The paper [Stratulat, 2012] presenting it received a *best paper award* at the Alan Turing Centenary Conference in 2012.

Certification of formula-based induction reasoning. Our experiments conducted on the JavaCard application, described in Chapter 5, with the SPIKE prover, presented in Chapter 4, have automatically generated some proof scripts consisting of thousands of steps that are time-consuming or very hard to be checked by humans. This motivated our research on the mechanical certification of the implicit induction proofs produced by SPIKE and, more general, of the formula-based induction reasoning using the Coq certification environment, as described in Chapter 6. This work was done in collaboration with Vincent Demange and Amira Henaien during their PhD period.

Connections between formula-based Noetherian induction reasoning and other kinds of first-order reasoning. In Chapter 7, we showed that the soundness conditions for a class of FOL_{ID} cyclic pre-proofs can be deduced from a set of ordering and derivability constraints issued from the analysis of the pre-proofs. The similarity between these ordering constraints and those occurring in formula-based proofs allowed us to extend the certification methodology for formula-based reasoning to certify FOL_{ID} cyclic proofs. In other direction, in Chapter 8 we identified similarities between reductive induction-based inference systems, as the implicit induction inference systems, and saturation-based inference systems, as the paramodulation- and resolution-based systems. For example, we showed that the derivations produced by these inference systems do not eliminate the minimal counterexamples. A methodology for building sound reductive induction-based systems has been adapted for saturation-based systems and used to improve some of the existing ones.

Case studies, computer experiments and software developments. The strengths and limits of our results have been illustrated by examples, non-trivial case studies and computer experiments. The most important software developments are the new version of the SPIKE prover and E-Cyclist (the extension of the FOL_{ID} Cyclist prover with the new checking method for the soundness of FOL_{ID} pre-proofs).

Part I

First-Order Noetherian Induction Proving

1

First-Order Instances of the Noetherian Induction Principle

Sommaire

1.1	**Basic notions**	**1**
1.2	**The term- and formula-based instances**	**3**
1.3	**Term-based Noetherian induction principles**	**5**
1.4	**Formula-based Noetherian induction principles**	**6**
1.5	**Conclusions**	**10**

Induction reasoning is helpful when establishing properties about programs and specifications built from recursively defined domains and data structures. The properties of interest are *inductive*, i.e., they are valid only in distinguished models, e.g., term-generated, standard or non-standard models as the Henkin models. However, it is hardly imaginable to perform induction reasoning exclusively with reasoning techniques working only for inductive models. Indeed, the current inductive theorem provers mix them with *deductive* techniques that can validate properties in all models.

This chapter, based on [Stratulat, 2012], gives an overview of the use of the Noetherian induction principle to build first-order proofs valid in term-generated models.

Structure of the chapter. The chapter has five sections. After the introduction of basic notions in Section 1.1, Section 1.2 defines the term- and formula-based instances of the Noetherian induction principle, adapted when reasoning on first-order logic. Sections 1.3 and 1.4 overview the term- and formula-based induction proof methods. The features of each of these methods are analysed by the means of different proofs of a very simple running example. The last section concludes.

1.1 Basic notions

Let $\mathcal{F}$ (resp. $\mathcal{P}$) be a ranked alphabet of function (resp. predicate) symbols and $\mathcal{V}$ a countable set of variables. $\mathcal{T}(\mathcal{F}, \mathcal{V})$ denotes the set of terms over $\mathcal{F}$ and $\mathcal{V}$, and $\mathcal{T}(\mathcal{F})$ its subset of ground terms. The expression $P(t_1, \ldots, t_n)$ is an *atom*, where $P \in \mathcal{P}$ is an n-ary predicate and $t_1, \ldots, t_n$ are terms. The set $\mathcal{A}(\mathcal{P}, \mathcal{F}, \mathcal{V})$ (resp. $\mathcal{A}(\mathcal{P}, \mathcal{F})$) denotes the set of atoms over $\mathcal{P}$, $\mathcal{F}$ and $\mathcal{V}$ (resp. $\mathcal{P}$ and $\mathcal{F}$). Let $\mathcal{L}$ represent a decidable set of first-order formulas over $\mathcal{A}$. We denote by Ax the *axioms* that build a specification, consisting of formulas from $\mathcal{L}$. New terms and formulas can be obtained by substitution operations, representing finite mappings between variables and terms, of the form $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, where x_i ($i \in [1..n]$) are distinct. A *renaming* substitution has all substituting terms as variables. In addition, if every substituting variable is the same as the substituted variable, the substitution is an *identity* substitution. Given a substitution σ and a term t (resp. formula ϕ), $t\sigma$ (resp. $\phi\sigma$), sometimes written (t, σ) or $t[\sigma]$ (resp. (ϕ, σ) or $\phi[\sigma]$), denotes the *instance* of t (resp. ϕ) by the substitution σ. Given two

substitutions σ_1 and σ_2, by $\psi(\sigma_1\sigma_2)$ we denote $(\psi\sigma_1)\sigma_2$, for any term or formula ψ; $\sigma_1\sigma_2$ means that σ_1 is *composed* with σ_2. We are referring to *ground* substitutions if the mapping terms are ground.

Semantically, given a first-order structure $\mathcal{S}$ with domain $\mathcal{D}$, we write $f^{\mathcal{S}}$ to denote the interpretation of the symbol $f \in \mathcal{F}$ in $\mathcal{S}$. The variables are interpreted by a *valuation* (total) function from variables of $\mathcal{V}$ to $\mathcal{D}$. It can be extended to non-variable terms as follows: if $t(s_1, \ldots, s_n)$ is a term such that $t^{\mathcal{S}} : D^n \to D$ is the interpretation of t, its interpretation is obtained by replacing each of its i) function symbol f by $f^{\mathcal{S}}$, and ii) variable by its valuation. Every n-ary predicate symbol from $\mathcal{P}$ is interpreted as an n-ary relation on D. We write $\mathcal{S} \models_\zeta \phi$ for a formula ϕ that is true in $\mathcal{S}$ using the valuation function ζ. Any *interpretation* $< \mathcal{S}, \zeta >$ satisfying Ax is a *model* of Ax. A formula ϕ is a *deductive consequence* of a set of formulas Φ, denoted by $\Phi \models \phi$, if ϕ holds in all models of Ax whenever ψ holds in all models of Ax, for any $\psi \in \Phi$. A formula is *deductively valid* w.r.t. Ax iff it is a deductive consequence of Ax.

Deductive relation. Let $\mathbb{P}(\mathcal{L})$ denote recursive sets over $\mathcal{L}$. The recursively enumerable *deductive relation* $\vdash \subseteq \mathbb{P}(\mathcal{L}) \times \mathcal{L}$ satisfies the properties:

- if $\phi \in Ax$, then $Ax \vdash \phi$,
- if $Ax \vdash \phi$ and $Ax \subseteq Ax'$, then $Ax' \vdash \phi$,
- if $Ax \vdash \phi$ and $Ax \cup \{\phi\} \vdash \phi'$, then $Ax \vdash \phi'$, and
- if $Ax \vdash \phi$, then $Ax \vdash \phi\sigma$, for any substitution σ.

In practice, the first-order (deductive) systems that implement the deductive relation are normally *sound*, i.e., $Ax \vdash \phi$ implies $Ax \models \phi$, and *complete*, i.e., $Ax \models \phi$ implies $Ax \vdash \phi$, for any formula ϕ.

Inductive relations. We are interested in the case when Ax has term-based (Herbrand) models, i.e., whose valuation functions transform variables into ground terms. A formula ϕ is an *inductive consequence* of the axioms if ϕ holds in all Herbrand models of Ax. The *inductive theory* of Ax consists of all inductive consequences of Ax. In general, it is neither decidable, nor semi-decidable. For various reasons which depend not only on the nature and form of the axioms but also on the user's intuition, monotonicity behavior and operational feasibility criteria [Wirth and Gramlich, 1994], only a non-empty subset of the Herbrand models of Ax is considered. For example, when Ax is a set of universally quantified Horn clauses with equality, it is convenient to reason on the unique *initial* (minimal) model of Ax. The choice of the model subset influences the way the variables are instantiated during the induction proofs.

In this chapter, we consider that such subset, denoted by $\mathcal{M}$, exists and it is already fixed before performing any induction reasoning. To simplify the presentation, we assume that all variables are universally quantified. A formula ϕ is a $\mathcal{M}$ - *consequence* (or just consequence) of a set of formulas Φ, denoted by $\Phi \models_{\mathcal{M}} \phi$, if $\mathcal{S} \models_\zeta \phi$ whenever $\mathcal{S} \models_\zeta \psi$, for any $\psi \in \Phi$ and model $< \mathcal{S}, \zeta >$ from $\mathcal{M}$. A formula ϕ is $\mathcal{M}$-*valid* (or just valid), denoted by $\models_{\mathcal{M}} \phi$ iff it is a consequence of Ax. The two notions of consequence relation may coincide for particular cases; for example, a positive clause ϕ is an inductive consequence of a set of universally quantified Horn clauses with equality iff ϕ is valid in their initial model [Gramlich, 2005].

A formula ϕ is *false*, denoted by $\not\models_{\mathcal{M}} \phi$, if it is not valid. Any false formula has (or contains) at least one false ground instance, called *counterexample*. It can be easily shown that, for any formula ϕ and set of formulas Ψ, if $\Psi \models_{\mathcal{M}} \phi$ and $\not\models_{\mathcal{M}} \phi$ then it exists a formula $\psi \in \Psi$ such that $\not\models_{\mathcal{M}} \psi$.

Sufficient completeness. The axioms define very often functions based on constructors that are sufficiently complete. In this case, the set $\mathcal{F}$ is split into defined function ($\mathcal{DF}$) and constructor ($\mathcal{C}$) symbols. In addition, the models from $\mathcal{M}$ are constructor models, i.e., whose valuation functions transform variables into ground constructor terms from $\mathcal{T}(\mathcal{C})$.

A function symbol $f \in \mathcal{DF}$ is *sufficiently complete* if any ground term of the form $f(\bar{t})$ is deductively equivalent to a constructor ground term from $\mathcal{T}(\mathcal{C})$, where $\bar{t}$ is a term vector of the form $(t_1, \ldots, t_n)$ with $t_i \in \mathcal{T}(\mathcal{C}), \forall i \in [1..n]$.

Induction orderings. The induction orderings are assumed to satisfy essential properties related to well-foundedness and stability under substitutions, as defined below. Well-foundedness implies the existence of minimal elements.

Lemma 1 *Let $(\mathcal{E}, <)$ be a well-founded poset. For any non-empty subset of $\mathcal{E}$, there is a minimal element.*

Proof By contradiction, assume that $\mathcal{E}$ has a non-empty subset $\mathcal{E}'$ without minimal elements, i.e., for each element of $\mathcal{E}'$ there is a smaller element in $\mathcal{E}'$. Let x_1 be an arbitrary element of $\mathcal{E}'$. Since it is not minimal, it exists an element x_2 of $\mathcal{E}'$ smaller than x_1. The same reasoning is performed on x_2, and so on, to finally build an infinite strictly decreasing sequence of elements of $\mathcal{E}'$. Contradiction because the ordering is well-founded. □

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 quasi-ordering is *total* if any two distinct elements are comparable.

In a first-order setting, the set $\mathcal{E}$ may contain an infinite number of elements that can be either ground terms or ground formulas. An effective reasoning can be pursued only on finite descriptions of them by the means of terms and/or formulas with variables. Hopefully, this reasoning can be projected and reused to the ground level when needed if the ordering is stable under substitutions.

An example of syntactic reduction ordering over terms is the *recursive path ordering* (rpo) [Dershowitz, 1982b, 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 $i \in [1..n]$, $a_i \sim_{rpo} b_i$. We refer to *multiset path ordering* (mpo) instead of rpo when the status of all function symbols is Mul. 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 there are two finite multisets X and Y such that $B = (A - X) \uplus Y$, $X \neq \emptyset$ and $\forall y \in Y$, $\exists x \in X, y < x$ holds, where $\uplus$ (resp., $-$) is the union (resp., difference) on multisets. In practice, X (resp., Y) is A (resp., B) after having deleted pairwisely the common elements. 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].

1.2 The term- and formula-based instances

As a running example, let us prove the conjecture $x+0 = x$, for all natural x, using the axioms $0+x = x$, for any natural x, and $S(x) + y = S(x + y)$, for any naturals x and y, that define the addition symbol '+' over the naturals based on the constructor symbols 0 and S. '=' is the only predicate symbol and the deductive system is based on equational logic [Baader and Nipkow, 1998]. '+' is sufficiently complete and the unique model from $\mathcal{M}$ is the initial model of the axioms since it fits well to reason over naturals.

Let n be an arbitrary natural, hence it is either 0 or a successor of another natural n'. In the first case, $0 + 0 = 0$ is deductively valid as an instance of the first axiom. The second case, $S(n') + 0 = S(n')$,

is deductively equivalent to $S(n'+0) = S(n')$ using the second axiom. The IH $n'+0 = n'$ can help to replace $S(n'+0)$ by $S(n')$ in order to obtain an identity and finish the proof.

The sound use of $n'+0 = n'$ can be argued by two different instances of the Noetherian induction principle, recalled below

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

according to the cases when the set $\mathcal{E}$ consists of ground terms or ground formulas. The underlying induction orderings over (vector of) terms and formulas are generically denoted by $<_t$ and $<_f$, respectively.

Term-based Noetherian induction. Let ϕ be a first-order formula defining a property to be checked for a non-empty set of term vectors $\mathcal{E}$. If $\bigcup_{\overline{k} \in \mathcal{E}, \overline{k} <_t \overline{m}} \{\phi(\overline{k})\} \models_{\mathcal{M}} \phi(\overline{m})$, for any term vector $\overline{m} \in \mathcal{E}$, then $\forall \overline{p} \in \mathcal{E}, \models_{\mathcal{M}} \phi(\overline{p})$.

Proof (soundness) By contradiction, if $\phi(\overline{m})$ is assumed to be false for some $\overline{m} \in \mathcal{E}$, we define the non-empty subset $\mathcal{E}' \subseteq \mathcal{E}$ representing all term vectors for which ϕ is false. By Lemma 1, $\mathcal{E}'$ has minimal term vectors, and let $\overline{m'}$ be such a term vector. The term-based Noetherian induction rule can be applied to prove the existence of a term vector from $\mathcal{E}'$ smaller than $\overline{m'}$, so contradiction. □

A well-known term-based Noetherian induction principle is the Peano induction principle.

Example 4 (Peano induction proof of $x+0=x$) *The IH $P(x')$ can be soundly used in the proof of $P(S(x'))$ because $x' < S(x')$, where $<$ is the 'smaller' relation over the naturals. In our example, $P(x)$ stands for $x+0=x$, so $n'+0=n'$ is the IH for proving $S(n')+0=S(n')$.*

On the other hand, the IH $n'+0=n'$ can also be legitimated by formula-based Noetherian induction principles.

Formula-based Noetherian induction. Let $\mathcal{E}$ be a non-empty set of first-order formulas. If for any formula $\delta \in \mathcal{E}, \bigcup_{\gamma \in \mathcal{E}, \gamma <_f \delta} \{\gamma\} \models_{\mathcal{M}} \delta$ then $\forall \rho \in \mathcal{E}, \models_{\mathcal{M}} \rho$.

Proof (soundness) The Noetherian induction principle presented in the introductory part is instantiated such that $\mathcal{E}$ consists of first-order formulas and the predicate ϕ is the identity relation, i.e., $\phi(x) = x$, for any formula $x \in \mathcal{E}$. □

Formulated into a 'Descente Infinie' setting, this principle states that a potentially infinite set of first-order formulas are true if for any false formula there is another formula which is also false but smaller w.r.t. the well-founded ordering. The proof of this statement is by contradiction: we assume a false formula in $\mathcal{E}$. We consider the non-empty subset set $\mathcal{E}'$ of $\mathcal{E}$ consisting of all the false formulas from $\mathcal{E}$. By Lemma 1, there exists a minimal false formula in $\mathcal{E}'$ for which there is no smaller false formula, so contradiction.

Example 5 *The formula-based Noetherian induction principle can be applied if the proof of $x+0=x$ has been generated using some reductive system such that, for any natural x', $x'+0=x'$ is smaller than $S(x'+0)=S(x')$ which is in turn smaller than $S(x')+0=S(x')$.*[6] *The set $\mathcal{E}$ consists of all formulas encountered in the proof script of $x+0=x$, i.e., $\{x+0=x, 0+0=0, S(x')+0=S(x'), S(x'+0)=S(x'), S(x')=S(x')\}$. The soundness proof uses a* reductio ad absurdum *reasoning at the ground level. By contradiction, we assume that $\mathcal{E}$ has a counterexample. Since the ordering is well-founded, there is a minimal counterexample of it. It can only be an instance of $x+0=x$ because the deductively valid formulas and the formulas deductively equivalent with but greater than $x+0=x$ cannot have minimal counterexamples. Let it be $n'+0=n'$, for some natural n'. n' should be of the form $S(n'')$ since $0+0=0$ is deductively true. In the proof script, $S(x)+0=S(x)$ is transformed into the smaller equality*

[6] Such an ordering over equalities can be the multiset extension of the mpo based on the increasing precedence over the function symbols 0, S, and $+$.

$S(x+0) = S(x)$, for any natural x, so $S(n''+0) = S(n'')$ is a smaller counterexample thanks to the 'stability under substitutions' property of the ordering. In the next step of the proof script, $S(x+0) = S(x)$ is transformed into an identity using $x+0 = x$, for any x. Instantiating x with n'', we conclude that $n''+0 = n''$ should be false. We get a contradiction since $n''+0 = n''$ is a counterexample of $x+0 = x$, smaller than $n'+0 = n'$.

1.3 Term-based Noetherian induction principles

Most of the term-based Noetherian induction principles promote *eager* induction, an approach that defines the IHs (sometimes long) before their use by means of (explicit) *induction schemas* [Aubin, 1979] usually resulted from the *recursion analysis* of recursively defined functions. Given a formula, an induction schema firstly identifies a subset of its variables to be instantiated, called *induction variables*, then defines the IHs as well as the induction conclusions as instances of the formula. The IHs associated to an induction conclusion are explicitly added to its conditions and are expected to participate in further developments of the proof. Some variables from the terms instantiating the induction variables are shared between the induction conclusions and their associated IHs.

A well-known example of 'induction schema'-based induction principle that fits well for constructor-based sorted specifications is the *structural induction* [McCarthy and Painter, 1967], which generalizes the Peano and mathematical inductions.

Theorem 1 (soundness of structural induction) *Let ϕ be a formula to be checked for the elements of a sort S. If $\forall f : S_1, \ldots, S_n \to S \in \mathcal{C}, \forall x_1, \ldots, \forall x_n, \{\phi(x_{i_1}) \cup \ldots \cup \phi(x_{i_k})\} \models_{\mathcal{M}} \phi(f(x_1, \ldots, x_n))$ then $\forall p \in S, \models_{\mathcal{M}} \phi(p)$, where the variables $x_{i_1}, \ldots, x_{i_k}$ are those variables among $x_1, \ldots, x_n$ that have the sort S.*

Proof The structural induction principle exploits the fact that x is structurally smaller than $f(\ldots, x, \ldots)$. The term-based induction principle is applied further. □

In turn, it is generalized by the *cover set induction* [Zhang *et al.*, 1988] which is inspired from the idea of [Boyer and Moore, 1979] according to which the induction schemas are built from the recursive definitions of the functions appearing in the conjecture to be proved. The cover set induction principle assumes that a sort is characterized, or covered, by a finite set of terms called (term) *cover set*. The cover set notion can be generalized to a set of term vectors that cover a product of sorts $S_1 \times S_2 \times \ldots$. Formally, $\{\overline{t_1}, \ldots, \overline{t_n}\}$ is a cover set of the product of sorts $\mathcal{E}$ if, for any formula ϕ, whenever $\not\models_{\mathcal{M}} \phi(\overline{u})$, for some term vector $\overline{u} \in \mathcal{E}$, it exists $j \in [1..m]$ and a substitution σ such that $\overline{t_j}\sigma \equiv \overline{u}$, where $\equiv$ is the (syntactical) identity relation.

Theorem 2 (soundness of cover set induction) *Let Ψ be a non-empty cover set $\{\overline{t_1}, \ldots, \overline{t_m}\}$ of the product of sorts $\mathcal{E}$ and ϕ a formula to be checked for the elements of $\mathcal{E}$. If $\bigcup_{\overline{k} \in \mathcal{E}, \overline{k} <_t \overline{t}} \{\phi(\overline{k})\} \models_{\mathcal{M}} \phi(\overline{t})$, for any term vector $\overline{t} \in \Psi$, then $\forall \overline{p} \in \mathcal{E}, \models_{\mathcal{M}} \phi(\overline{p})$.*

Proof By contradiction, assume that $\exists \overline{p'} \in \mathcal{E}$ such that $\models_{\mathcal{M}} \phi(\overline{p'})$ is false. We consider $\mathcal{E}' \subseteq \mathcal{E}$ defined as $\{\overline{p} \mid \overline{p} \in \mathcal{E}, \models_{\mathcal{M}} \phi(\overline{p}) \text{ is false}\}$. $\mathcal{E}'$ is not empty since $\overline{p'} \in \mathcal{E}'$. By the well-foundedness property of $<_t$ and Lemma 1, there is a minimal term vector $\overline{u} \in \mathcal{E}'$ such that $\not\models_{\mathcal{M}} \phi(\overline{u})$. By the definition of the term cover set, it exists $j \in [1..m]$ and a substitution σ such that $\overline{t_j}\sigma \equiv \overline{u}$. On the other hand, $\bigcup_{\overline{k} \in \mathcal{E}, \overline{k} <_t \overline{t_j}} \{\phi(\overline{k})\} \models_{\mathcal{M}} \phi(\overline{t_j})$. By the 'stability under substitutions' property of $<_t$, we have $\bigcup_{\overline{k} \in \mathcal{E}, \overline{k}\sigma <_t \overline{t_j}\sigma} \{\phi(\overline{k}\sigma)\} \models_{\mathcal{M}} \phi(\overline{t_j}\sigma)$. Since $\not\models_{\mathcal{M}} \phi(\overline{t_j}\sigma)$, it exists $\overline{k'} \in \mathcal{E}$ such that $\overline{k'} <_t \overline{u}$ and $\not\models_{\mathcal{M}} \phi(\overline{k'})$. Therefore, $\overline{k'} \in \mathcal{E}'$. Contradiction with the minimality of $\overline{u}$. □

The main shortcomings of the schemata-based approaches are related to the management of the IHs, when: i) the generated IHs do not contribute to the proof, and ii) the IHs that are required at some point of the proof are not yet generated or impossible to be defined with the 'recursion analysis' method. Its

main advantage is the *local scope* of the induction ordering, at the level of the induction schema. This allows for flexibility in the ordering management during a proof; the ordering constraints are checked only when defining the induction schemas and, on the other hand, a different induction ordering may be used for each induction schema. Moreover, the schemata-based induction can be easily integrated into sequent-based deductive systems in terms of sound inference rules.

Another shortcoming is the treatment of mutual induction which cannot be directly performed because the induction conclusion and its attached IHs are instances of the *same* formula. However, several partial solutions have been proposed. Boyer and Moore [Boyer and Moore, 1988a] build the induction schema from a new function that calls the mutually defined functions according to the value of an extra argument. Sometimes, the conjecture should be strengthened or auxiliary lemmas about the mutually defined functions should be provided by the users. A more automatic solution has been provided by Kapur and Subramaniam [Kapur and Subramaniam, 1996] to deal with a class of mutually recursive functions that can transform the mutual recursion into simple recursion by unrolling the cover sets and expanding the function definitions. The multi-predicate induction schemas proposed by Boulton and Slind [Boulton and Slind, 2000] have one predicate for each of the mutually recursive functions and avoid the need for expanding the functions into a single function. However, if the conjecture embeds different recursive function symbols the induction schemas have to be combined [Boyer and Moore, 1979].

1.4 Formula-based Noetherian induction principles

Inductionless induction, also known as *proof by consistency*, is the first proof method integrating formula-based Noetherian induction. Proposed by Musser in [Musser, 1980], it uses the saturation-based Knuth-Bendix's completion algorithm [Knuth and Bendix, 1970] to prove inductive properties. The method can prove a set of equalities as consequences of a consistent set of equality axioms by i) adding them to the axioms, ii) orienting the new set of equalities into rewrite rules, and ii) showing their consistency if the completion algorithm saturates, i.e., until no new[7] equality is generated. As time went by, the method has been improved [Huet and Hullot, 1980, Dershowitz, 1982a, Fribourg, 1986, Jouannaud and Kounalis, 1986, Kapur *et al.*, 1986, Bachmair, 1988, Küchlin, 1989]. For an overview of inductionless induction, the reader may consult [Comon, 2001, Comon and Nieuwenhuis, 2000].

By clearly separating the axioms from the formulas to be proved, the *implicit induction* inference systems are reductive procedures, many of them being rewrite-based and saturation-free. Initially, Reddy [Reddy, 1990] proposed a proof method called *term-rewriting induction*, implemented by an inference system which computes *formula cover sets*. They are issued from the instantiation of some variables of a formula with a term cover set associated to the product of their sorts defined in terms of *cover substitutions*. Each formula instance is reduced afterwards with rewrite rules from the axioms; in this way, whenever the formulas from the formula cover set are valid, the covered formula is also valid. Bachmair [Bachmair, 1988] showed that formula cover sets are fundamental for the proofs by consistency. This is also true for the proofs by implicit induction. In the following, we refine formula cover sets as (general) cover sets and *strict* cover sets. Formally, $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ is a set of cover substitutions of a formula $\phi(\overline{x})$ if the set of mapping term vectors built for each substitution from Σ is a term cover set of the (product) sort of $\overline{x}$. A set of formulas $\{\phi_1, \ldots, \phi_n\}$ is a formula cover set (resp. strict formula cover set) of a formula ϕ built on the set of cover substitution $\{\sigma_1, \ldots, \sigma_n\}$ if $\phi_i \models_{\mathcal{M}} \phi\sigma_i$ and $\phi_i \leq_f \phi\sigma_i$ (resp. $\phi_i <_f \phi\sigma_i$), for any $i \in [1..n]$.

The term-rewriting method does not need to be *refutationally complete*, as required by [Bachmair, 1988], and the specifications may not be *ground confluent* in order to prove inductive properties.[8] Kounalis and Rusinowitch [Kounalis and Rusinowitch, 1990b] went even further and proposed a completion-free induction technique based on *test sets* [Kapur *et al.*, 1986].

To simplify the rest of the presentation, we will denote the ordering over formulas $<_f$ by $<$ and refer to (strict) formula cover sets as (strict) cover sets, if otherwise stated.

[7] w.r.t. some redundancy criteria.
[8] However, these properties are required to *refute* conjectures.

Theorem 3 (soundness of term-rewriting induction) *Let* $\{\phi\sigma_1, \ldots, \phi\sigma_n\}$ *be a set of instances of the formula* $\forall \overline{x} \in \mathcal{E}, \phi(\overline{x})$ *built from the cover-substitutions* $\sigma_1, \ldots, \sigma_n$*, respectively. If* $\forall i \in [1..n], \bigcup_{\phi\theta < \phi\sigma_i} \phi\theta \models_{\mathcal{M}} \phi\sigma_i$ *then* $\forall \overline{x} \in \mathcal{E}, \models_{\mathcal{M}} \phi(\overline{x})$.

Proof By absurd, assume that $\forall i \in [1..n], \bigcup_{\phi\theta < \phi\sigma_i} \phi\theta \models_{\mathcal{M}} \phi\sigma_i$ but it exists $\overline{u} \in \mathcal{E}$ such that $\not\models_{\mathcal{M}} \phi(\overline{u})$. We consider $\mathcal{E}'$ defined as $\{\phi\theta \mid \not\models_{\mathcal{M}} \phi\theta\}$. $\mathcal{E}'$ is not empty since $\phi(\overline{u}) \in \mathcal{E}'$. Since $<$ is well-founded, there is a minimal instance $\phi\tau$ in $\mathcal{E}'$, by Lemma 1. Also, $\sigma_1, \ldots, \sigma_n$ are cover-substitutions for ϕ, so there exist $j \in [1..n]$ and a substitution ϵ such that $\phi\tau \equiv \phi\sigma_j\epsilon$ and $\not\models_{\mathcal{M}} \phi\sigma_j\epsilon$. Since $\bigcup_{\phi\theta < \phi\sigma_i} \phi\theta \models_{\mathcal{M}} \phi\sigma_i$ for all $i \in [1..n]$, this is also true for j, so $\bigcup_{\phi\theta < \phi\sigma_j} \phi\theta \models_{\mathcal{M}} \phi\sigma_j$. By the 'stability under substitutions' of $<$, we have $\bigcup_{\phi\theta\epsilon < \phi\sigma_j\epsilon} \phi\theta\epsilon \models_{\mathcal{M}} \phi\sigma_j\epsilon$. On the other hand, $\not\models_{\mathcal{M}} \phi\sigma_j\epsilon$, so there is a substitution θ' such that $\not\models_{\mathcal{M}} \phi\theta'\epsilon$ and $\phi\theta'\epsilon < \phi\sigma_j\epsilon (\equiv \phi\tau)$. Contradiction with the minimality of $\phi\tau$. □

The term-rewriting induction principle cannot directly perform mutual induction reasoning because all the involved formulas are instances of only one formula. It has been superseded by the implicit induction principle, as suggested in [Kounalis and Rusinowitch, 1990b] and formally presented in [Bronsard *et al.*, 1994].

Theorem 4 (soundness of implicit induction) *Let* $\mathcal{E}$ *be a set of formulas and assume that for any formula* $\delta \in \mathcal{E}$, $\bigcup_{\gamma \in \mathcal{E}, \gamma < \delta} \{\gamma\} \models_{\mathcal{M}} \delta$*. Also, let* ϕ *be a formula to be checked for a non-empty set* $\mathcal{S}$ *of term vectors. If* $\forall \overline{p} \in \mathcal{S}, \phi(\overline{p}) \in \mathcal{E}$ *then* $\forall \overline{p} \in \mathcal{S}, \models_{\mathcal{M}} \phi(\overline{p})$.

Proof By the formula-based Noetherian induction principle, any formula δ from $\mathcal{E}$ holds. Since $\forall \overline{p} \in \mathcal{S}, \phi(\overline{p}) \in \mathcal{E}$, then $\forall \overline{p} \in \mathcal{S}, \models_{\mathcal{M}} \phi(\overline{p})$. □

In practice, the implicit induction technique is applied on the set $\mathcal{E}$ of all instances of the formulas encountered in a proof derivation. The formula $\phi(\overline{p})$ is one of its initial conjectures and the ordering constraints from the relation $\forall \delta \in \mathcal{E}, \bigcup_{\gamma \in \mathcal{E}, \gamma < \delta} \{\gamma\} \models_{\mathcal{M}} \delta$ are guaranteed by *reductive* inference systems. They consist of *inference rules* representing transitions between *states* consisting of pairs of sets of formulas of the form (E, H), where E are *conjectures* and H are *premises*. By applying an inference rule, one of the conjectures, called *current 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. A *derivation* is a successive application of inference rules. A *proof* of a set of formulas E^0 produced with the inference system J is a finite $n+1$-state derivation of the form $(E^0,\ \emptyset) \vdash_J (E^1,\ H^1) \vdash_J \ldots \vdash_J (\emptyset,\ H^n)$.

An inference system is *sound* if the minimal counterexamples are *persistent* in any derivation, i.e., whenever the current conjecture has a minimal counterexample, an equivalent one exists in the set of conjectures from a future state.

Theorem 5 *Let* I *be a sound inference system. For any proof* $(E^0,\ \emptyset) \vdash_I \ldots \vdash_I (\emptyset,\ H^n)$*, we have* $\models_{\mathcal{M}} E^0$.

Proof By contradiction, assume that E^0 has a false formula, hence a counterexample. By Lemma 1, in the set $\mathcal{E}$ there exists a minimal counterexample ϕ. Since I is sound, this counterexample is persistent and should be among the ground instances of the conjectures from the last state of the proof. Contradiction, since the proof finishes with an empty set of conjectures. □

A simple sound inference system is I:

GENERATE: $(E \cup \{\phi\},\ H) \vdash_I (E \cup \Psi,\ H \cup \{\phi\})$,
where Ψ is a strict cover set of ϕ.
SIMPLIFY: $(E \cup \{\phi\},\ H) \vdash_I (E \cup \Phi,\ H)$,
if $(E \cup \Phi \cup H)_{\leq \phi} \models_{\mathcal{M}} \phi$.

By $\Psi_{\leq \psi}$ (resp. $\Psi_{< \psi}$) are denoted the instances of formulas from Ψ that are smaller than or equal to (resp. strictly smaller than) ψ. The GENERATE rule replaces the current conjecture with a strict cover

set of it, then adds it to the set of premises. SIMPLIFY replaces ϕ from the state $(E \cup \{\phi\}, H)$ with Φ if ϕ is a consequence of the set of IHs $(E \cup \Phi \cup H)_{\leq\phi}$.

Lemma 2 *The premises from any I-derivation starting with an empty set of premises do not have minimal counterexamples.*

Proof Let us notice that the premises from any derivation that starts with an empty set of premises are added exclusively by GENERATE. By contradiction, assume that GENERATE applies on a current conjecture ϕ that has a minimal counterexample $\phi\tau$. By definition, the strict cover set Ψ for ϕ has a formula γ that covers $\phi\tau$, i.e., there is a cover substitution σ and another substitution ϵ such that $\phi\sigma\epsilon \equiv \phi\tau$ and $\bigcup_{\gamma<\phi\sigma}\{\gamma\} \models_{\mathcal{M}} \phi\sigma$. Thanks to the 'stability under substitutions' property of $<$, we have $\bigcup_{\gamma\epsilon<\phi\sigma\epsilon}\{\gamma\epsilon\} \models_{\mathcal{M}} \phi\sigma\epsilon$. On the other hand, $\not\models_{\mathcal{M}} \phi\sigma\epsilon$. Therefore, there is a ground instance from Ψ which is a counterexample smaller than $\phi\tau$. Contradiction. □

Theorem 6 (Soundness of I) *I is sound for any derivation starting with an empty set of premises.*

Proof Assume that a conjecture ϕ has a minimal counterexample $\phi\tau$. GENERATE cannot be applied on ϕ, by Lemma 2. If SIMPLIFY is applied to ϕ in the state $(E \cup \{\phi\}, H)$, we have $(E \cup \Phi \cup H)_{\leq\phi\tau} \models_{\mathcal{M}} \phi\tau$. Again by Lemma 2, H cannot have minimal counterexamples. Therefore, a minimal counterexample equivalent to $\phi\tau$ should exist in the conjectures $E \cup \Phi$ from the next state. □

I is an inference system that abstracts the computation, hence it cannot be used in practice. Its main role is to capture the induction reasoning by defining the formulas that can be used as IHs during a proof. Its concrete implementations show *how* the current conjecture is transformed, by the means of adequate reasoning techniques that may use the IHs defined by the implemented (abstract) inference rules. In addition, any such concrete implementation of I is sound since I has been shown sound by Theorem 6. The equality $x + 0 = x$ from the running example can be proved with the inference system I_i that allows to reason on equalities containing natural variables:

GENNAT (G): $(E \cup \{\phi\},\ H) \vdash_{I_i} (E \cup \{\phi_1, \phi_2\},\ H \cup \{\phi\})$,
  where ϕ has a natural variable that is instantiated by 0 and $S(x')$ and x' is a fresh variable;
  ϕ_1, ϕ_2 are the result of rewriting the instances of ϕ with axioms.
SIMPEQ (S): $(E \cup \{\phi\},\ H) \vdash_{I_i} (E \cup \Phi,\ H)$,
  if either i) ϕ is a tautology; in this case Φ is empty;
  or, ii) ϕ is rewritten to ψ with rewrite rules from $Ax \cup (E \cup \Phi \cup H)_{\leq\phi}$; in this case, Φ is $\{\psi\}$.

GENNAT builds a strict cover set of an equality integrating a natural variable by firstly replacing the variable by 0 and the successor of a fresh natural variable, then rewriting the two instances by the rewrite rules resulted from orienting the axioms from left to right whenever the lhs is greater than the rhs. The rewriting results are stored as new conjectures and the equality as a new premise. Therefore, it implements GENERATE. On the other hand, SIMPEQ implements SIMPLIFY. It either deletes the tautologies or performs rewrite operations based on the same ordering over equalities mentioned in the footnote 6, and on the IHs defined by SIMPLIFY.

Example 6 (I_i-proof of $x + 0 = x$) *Let $0 + x \to x$ and $S(x) + y \to S(x + y)$ be the two rewrite rules resulting from orienting from left to right the axioms defining '+'. The equality $x+0 = x$ can be proved with I_i, as follows: $(\{\underline{x + 0 = x}\}, \emptyset) \vdash^{G}_{I_i} (\{\underline{0 = 0}, S(x'+0) = S(x')\}, \{x+0 = x\}) \vdash^{S}_{I_i} (\{\underline{S(x' + 0) = S(x')}\}, \{x+0 = x\}) \vdash^{S}_{I_i} (\{\underline{S(x') = S(x')}\}, \{x+0 = x\}) \vdash^{S}_{I_i} (\emptyset, \{x+0 = x\})$. In the derivation, the current conjectures from every state are underlined. The induction reasoning is performed during the second last* SIMPEQ *application: the instance $x' + 0 = x'$ of the premise $x + 0 = x$ is applied as IH in order to reduce $S(x' + 0) = S(x')$ to the identity $S(x') = S(x')$. Since I_i is an instance of I, it is sound, so $x + 0 = x$ is valid, by Theorem 5.*

We present a more effective inference system, denoted by I_s, that instantiates I. It integrates the narrowing-based inference rule CONJSUP:

ConjSup (Cs): $(E \cup \{\phi\},\ H) \vdash_{I_s} (E \cup \cup_i^n \{\phi_i\},\ H \cup \{\phi\})$,
where $\phi_i = \phi[r_i]_p\sigma_i$ for any rewrite rule $l_i \to r_i$ from
Ax such that $\sigma_i = mgu(\phi|p, l_i)$ and $\phi|p$, with $i \in [1..n]$

The narrowing operation is based on unification. $mgu(s,t)$ denotes the *most general unifier* between s and t, for any terms s and t, i.e. the most general substitution σ such that $s\sigma \equiv t\sigma$.

ConjSup is adapted from [Comon and Nieuwenhuis, 2000] to perform *conjecture superposition* on a set of axioms that is saturated under superposition and equality reasoning.[9] It firstly chooses from an equality ϕ one of the non-variable subterms $\phi|p$ at a position p, then unifies it with the left hand sides of all rewrite rules from the axioms. Any time the unification process is successful, the subterm is replaced by the corresponding unification instance of the right-hand side of the rewrite rule and the resulted equality becomes a new conjecture. If the set of new conjectures is not empty, the current conjecture is saved as premise. $\phi[r_i]_p$ indicates that ϕ has the subterm r_i at position p. In [Stratulat, 2005, Stratulat, 2007], it has been shown that $\cup_i^n\{\phi_i\}$ from similar narrowing-based inference rules is a strict cover set of ϕ, as it is the case for ConjSup.

Example 7 (I_s-proof of $x + 0 = x$) *The proof starts by applying the rule* ConjSup *on the subterm* $x + 0$ *of* $x + 0 = x$. *Since* $x + 0$ *unifies with the left-hand sides of the two axioms defining '+', the new conjectures are* $0 = 0$ *and* $S(x' + 0) = S(x')$, *where* x' *is a fresh variable.* $x + 0 = x$ *is added to the premises, to finally obtain a result similar to that of* GenNat *in the precedent* I_i*-proof. The rest of the proof can be successfully done as for the* I_s*-proof if* I_s *integrates a rule like* SimpEq.

In practice, ConjSup may generate less new conjectures than GenNat. For example, if $S(u) + 0 = S(u)$ is the current conjecture, ConjSup will generate the singleton $\{S(u+0) = S(u)\}$, which is equivalent to a rewrite step, while GenNat yields the set of conjectures $\{S(0+0) = S(0), S(S(u+0)) = S(S(u))\}$.

Different sound abstract reductive systems exist in the literature. I is very similar to 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, 1990b, Bronsard and Reddy, 1991, Bouhoula *et al.*, 1995]. A very general inference system was proposed in [Stratulat, 2001], based on the notion of *contextual* cover set, that generalizes those of cover set and strict cover set. It is conducted by a methodology to build sound implicit induction procedures using the compositional properties of contextual cover sets. The methodology also allowed to represent saturation-based inference systems as instances of the general inference system [Stratulat, 2005, Stratulat, 2007]. It witnesses that the implicit induction and saturation-based procedures share the same logic. This is not surprising since Bachmair [Bachmair, 1988] and Reddy [Reddy, 1990] already shown that the set of critical pairs generated by completion can build (strict) cover sets.

When dealing with equality reasoning, the reductive constraints between the current and new conjectures can be implicitly satisfied by the reductive inference rules if the equational specifications are represented in terms of rewrite systems and the IHs are orientable, as for the inductionless induction methods. Various solutions have been proposed to partially weaken the constraints related to IHs. In [Stratulat, 2008], the proposed method allows for *relaxed* rewriting [Bouhoula *et al.*, 1995] to deal with unorientable IHs by integrating explicit induction schemas when building cover sets. It covers the *term* [Reddy, 1990], *ordered* [Dershowitz and Reddy, 1993], *enhanced* and *incremental* [Aoto, 2006] rewriting induction procedures.

Mutual recursion represents a form of recursion for which data types or functions are mutually defined. It is ubiquitous in computer science. Crucial data structures like graphs, forests of trees and some programming language constructions, as the set of syntax rules using the BNF notation, can be naturally represented as mutually recursive data types. On the other hand, the mutually recursive functions, very common in functional programming, can code effectively algorithms based on (mutually) recursive

[9] More about the links between superposition calculus and induction reasoning can be found in Chapter 8.

data structures. *Mutual induction* is the most natural induction technique to reason on datatypes and functions defined using mutual recursion.

The formula-based Noetherian induction reasoning can involve instances of *different* formulas, which makes easy the management of mutual induction [Stratulat, 2010]. As shown below, any term-based Noetherian induction principle can be represented as formula-based one but the instantiation constraints are so strong that the ability to perform mutual induction is lost. This instantiation result argues a certain resemblance between (parts of) implicit and conventional induction proofs, firstly advocated in Musser's paper [Musser, 1980]. For example, the I-proof of the running example is very similar to the term-based version; as for an explicit induction schema, GENERATE instantiates variables and adds the current conjecture in the set of premises, but the resemblance stops here. [Sprenger and Dam, 2003] and [Brotherston and Simpson, 2011] include similar instantiation results. Also, the formula-based Noetherian induction procedures can perform *lazy* induction such that the IHs are provided by request. For example, during any of the proofs of $x + 0 = x$ with formula-based Noetherian induction methods, the smaller instances of previous conjectures to be applied as IH are needed to be known only at the moment of their application.

In some cases, the implicit induction proofs are more automatic than those based on conventional induction [Bouhoula and Rusinowitch, 1995], in other cases the contrary happens [Kapur and Subramaniam, 1996]. Further analyses and comparisons have been conducted in [Garland and Guttag, 1988, Jouannaud and Kounalis, 1989, Kapur and Zhang, 1994, Naidich, 1996, Comon, 2001, Wirth, 2005].

The following theorem is important from both theoretical and practical points of view.

Theorem 7 *Any term-based Noetherian induction principle can be represented as a formula-based Noetherian induction principle.*

Proof Let us consider a term-based Noetherian induction principle that proves the validity of a formula ϕ for all term vectors from a non-empty set $\mathcal{E}$, i.e., if for any term vector $\overline{m} \in \mathcal{E}$, $\bigcup_{\overline{k}\in\mathcal{E},\overline{k}<_t\overline{m}}\{\phi(\overline{k})\} \models_{\mathcal{M}} \phi(\overline{m})$ then $\forall\overline{p} \in \mathcal{E}, \models_{\mathcal{M}} \phi(\overline{p})$. Let $\mathcal{E}'$ be the set $\{\phi(\overline{p}) \mid \overline{p} \in \mathcal{E}\}$. The equivalent formula-based Noetherian induction principle can be stated as: if for any formula $\phi(\overline{m}) \in \mathcal{E}'$, $\bigcup_{\phi(\overline{k})\in\mathcal{E}',\phi(\overline{k})<_f\phi(\overline{m})}\{\phi(\overline{k})\} \models_{\mathcal{M}} \phi(\overline{m})$ then we have $\models_{\mathcal{M}} \phi(\overline{p})$, $\forall\phi(\overline{p}) \in \mathcal{E}'$, where $\phi(\overline{k}) <_f \phi(\overline{m})$ is defined as $\overline{k} <_t \overline{m}$. □

Corollary 1 *Any term-based Noetherian induction proof can be justified with formula-based induction arguments.*

In practice, the term-based Noetherian induction reasoning can therefore be *directly* integrated by formula-based Noetherian induction inference systems. Unfortunately, the opposite holds only for particular cases.

Theorem 8 *The formula-based Noetherian induction principle can be represented as a term-based Noetherian induction principle if $\mathcal{E}$ consists only of instances of a same formula.*

Proof The formula-based Noetherian induction principle that proves the validity of a non-empty set of instances of some formula ϕ is formalized as following: if for any formula $\phi(\overline{m}) \in \mathcal{E}$, $\bigcup_{\phi(\overline{k})\in\mathcal{E},\phi(\overline{k})<_f\phi(\overline{m})}\{\phi(\overline{k})\} \models_{\mathcal{M}} \phi(\overline{m})$ then we have $\models_{\mathcal{M}} \phi(\overline{p})$, $\forall\phi(\overline{p}) \in \mathcal{E}$. Let $\mathcal{E}'$ be the set $\{\overline{p} \mid \phi(\overline{p}) \in \mathcal{E}\}$. The equivalent term-based Noetherian induction principle can be stated as: if for any term vector $\overline{m} \in \mathcal{E}'$, $\bigcup_{\phi(\overline{k})\in\mathcal{E}',\overline{k}<_t\overline{m}}\{\phi(\overline{k})\} \models_{\mathcal{M}} \phi(\overline{m})$ then we have $\models_{\mathcal{M}} \phi(\overline{p})$, $\forall\overline{p} \in \mathcal{E}'$, where $\overline{k} <_t \overline{m}$ is defined as $\phi(\overline{k}) <_f \phi(\overline{m})$. □

1.5 Conclusions

We have given an overview of the Noetherian induction proof methods for first-order logic and term-generated models. We compared different instances of the Noetherian induction principle and inference systems based on them. A crucial step was to abstract the computation in order to distile the induction

reasoning from the implementation details. Some examples of concrete implementations instantiating the abstract inference systems are presented, but most of the problems and challenges that face current implementations are not discussed here (see, e.g., [Gramlich, 2005] for an overview).

2

A Cyclic and Non-reductive Formula-based Noetherian Induction Proof Method

Sommaire

2.1	**The inference system**	**13**
2.2	**The DRaCuLa proof strategy**	**14**
2.3	**Building cycles**	**17**
2.4	**Managing mutual induction**	**21**
2.5	**Representing implicit induction proofs as cyclic proofs**	**22**
2.6	**Conclusions**	**24**

This chapter presents a formula-based Noetherian induction proof method that is cyclic and non-reductive, published in [Stratulat, 2012].

Structure of the chapter. The chapter has six sections. Sections 2.1 and 2.2 present an inference system and proof strategy, respectively, for building cyclic and non-reductive formula-based proofs. The definition of a cycle, the soundness proof of the inference system and the connections between reductive and non-reductive proofs are given in Section 2.3. A full example of proof using this method and requiring mutual induction reasoning is presented in Section 2.4. Section 2.5 compares implicit proofs and proofs built with an implementation of the method and the DRaCuLa strategy in the SPIKE theorem prover [Bouhoula *et al.*, 1992, Stratulat, 2001, Barthe and Stratulat, 2003] for the validation of a non-trivial application. Section 2.6 concludes.

2.1 The inference system

The features of formula- and term-based Noetherian induction proof techniques mutually complement each other. In the following, we propose a new induction proof method which preserves the advantages of conventional and implicit induction techniques. The heart of the method is the DRaCuLa strategy, designed for performing:

- formula-based 'Descente Infinie' / Noetherian induction,
- Rarefied ordering constraints by reductive-free induction,
- Customized term-based Noetherian induction, and
- Lazy and mutual induction.

A DRaCuLa-based inference system consists of a set of inference rules representing transitions between sets of conjectures. We introduce the inference system D made of three non-reductive abstract inference rules:

DEDUCTION (D): $E \cup \{\phi\} \vdash_D E \cup \Phi$,
if $\Phi \models \phi$.
SPLIT (S): $E \cup \{\phi\} \vdash_D E \cup \Phi$,
if Φ is a cover set of ϕ with at least two elements.
INDUCTION (I): $E \cup \{\phi\} \vdash_D E \cup \Phi$,
if $\Phi \cup \Psi \models_{\mathcal{M}} \phi$ and Ψ is a non-empty set of checked IHs.

DEDUCTION (resp. SPLIT) replaces the current conjecture by new formulas for which it is a deductive consequence (resp. by one of its cover sets). In addition, SPLIT is intended to deal with variable instantiations during a proof derivation, hence the requirement for the cover set to have at least two elements. INDUCTION treats the case when IHs are needed to build the new conjectures, but it can be applied only when the IHs are checked according to the DRaCuLa strategy.

It is assumed that each conjecture ϕ has attached a *history* consisting of a list of conjecture instances represented as pairs (ϕ^i, σ^i) and involved in producing ϕ. Formally, it is denoted by $\xrightarrow[...(\phi^i,\sigma^i)(\phi^{i+1},\sigma^{i+1})...]{} \phi$, where the history is shown under the horizontal arrow pointing to ϕ. The time flows from left to right, for example ϕ^i was created before ϕ^{i+1}. The conjectures from the history of ϕ are its *ancestors*. Each instantiating substitution from the history is an *identity* substitution, excepting when SPLIT is applied. For this case, the cover substitution involved in producing ϕ is considered instead. The *offsprings* of a formula ϕ are all formulas from a derivation having ϕ in the history. In addition, ϕ has attached a set of IHs to be checked before INDUCTION can be applied.

2.2 The DRaCuLa proof strategy

When a proof starts, the history and the set of attached IHs for each conjecture from the set of initial conjectures E^0 are empty. The DRaCuLa strategy mingles proof development, as implemented by the procedure 'Develop' (see Algorithm 1), with IH checking performed by the function 'Check' (see Algorithm 2). Algorithm 1 applies D-inference rules one by one starting from E^0. The application of any INDUCTION rule $E \cup \{\phi\} \vdash_D^I E \cup \Phi$ attaches the set Ψ of IHs to ϕ and is delayed until all IHs from Ψ are checked. In this case, ϕ is in *stand-by* and no other rule rule can be applied to it as long as the set of attached IHs is not completely checked. Any IH is an instance (ϕ^1, δ), where ϕ^1 is a previous conjecture encountered in the derivation. Two cases may arise for successfully checking (ϕ^1, δ): i) if ϕ^1 was already proved, i.e., ϕ^1 has no offsprings in E, and ii) if (ϕ^1, δ) is part of a *cycle* built from ϕ and other conjectures in stand-by. Otherwise, the proof keeps developing other conjectures, hoping that the newly added conjectures build cycles that successfully check (ϕ^1, δ). The proof process successfully finishes when the set of conjectures becomes empty. Also, the proof development may be blocked if all conjectures from the current state are in stand-by. In practice, the deadlock can be avoided since other rules like SPLIT and DEDUCTION can be applied on the stand-by conjectures if the set of attached IHs is reinitialized. Finally, a proof may run infinitely if a crucial IH cannot be checked.

Definition 1 (n-cycle) *Let us consider n conjectures $\phi_1, \ldots, \phi_n$ such that, for each $i \in [1..n]$, ϕ_i is explicitly represented with its history chunk involved in the cycle as $\xrightarrow[(\phi_i^1,\sigma_i^1)...(\phi_i^{m_i},\sigma_i^{m_i})]{\theta_i=\sigma_i^1...\sigma_i^{m_i}} \phi_i$ and has attached the IH $(\phi^1_{(i+1)\ mod\ n}, \delta_{(i+1)\ mod\ n})$. The n conjectures form a n-cycle if $\phi^1_{(i+1)\ mod\ n}\delta_{(i+1)\ mod\ n}$ is smaller than $\phi_i^1\theta_i$, for any $i \in [1..n]$.*

Algorithm 1 Develop(E): applies successively D-inference rules to build a proof of the conjectures from E

while E is not empty **do**
  IHS := $\{\psi \mid \exists \phi \in E$ that attached the unchecked IH $\psi\}$
  choose $\phi \in E$ with no unchecked IHs
  choose an inference rule to apply on ϕ
  if the set of IHs attached to ϕ is not empty **then**
    the chosen inference rule should be INDUCTION
  else
    if INDUCTION was chosen with the set Ψ of IHs **then**
      assign as unchecked and attach all IHs from Ψ to ϕ
      IHS := IHS $\cup$ Ψ
    end if
  end if
  if the chosen inference rule is INDUCTION **then**
    ok_IHs := $\emptyset$
    for all $(\phi_i^1, \delta_i) \in$ IHS **do**
      offsprings_ϕ_i^1 := $\{\phi' \mid \xrightarrow[hist]{} \phi' \in E$ and $\phi_i^1 \in hist\}$
      if offsprings_ϕ_i^1 is empty **then**
        ok_IHs := ok_IHs $\cup$ $\{(\phi_i^1, \delta_i)\}$ $\{\phi_i^1$ *was proved*$\}$
      end if
    end for
    ok_IHs := Check(IHS $\setminus$ ok_IHs, E) $\cup$ ok_IHs
    mark all IHs from ok_IHs as checked
  end if
  if all attached IHs to ϕ are checked **then**
    apply the chosen inference rule $E \cup \{\xrightarrow[hist]{} \phi\} \vdash_D E \cup \Phi$
    attach to each conjecture from Φ an empty set of IHs
    if inference rule is SPLIT **then**
      update any cover instance $\phi' \equiv (\phi, \sigma)$ to $\xrightarrow[hist;(\phi,\sigma)]{} \phi'$
    else
      update each $\phi' \in \Phi$ to $\xrightarrow[hist;(\phi,\sigma_{id})]{} \phi'$, where
      σ_{id} is the identity substitution instantiating all variables of ϕ.
    end if
    $E := (E \setminus \{\phi\}) \cup \Phi$
  end if
end while

Algorithm 2 Check(IHS, E): identifies cycles based on IHs from IHS and conjectures from E

Ensure: return all IHs from the identified cycles
  ok_IHs := $\emptyset$
  repeat
    find a non-empty list of n conjectures $\phi_1, \ldots, \phi_n$ from E, $\xrightarrow[\ldots(\phi_i^1,\sigma_i^1)\ldots(\phi_i^{m_1},\sigma_i^{m_1})]{}\ \phi_i$, $i \in [1..n]$
    if $(\phi_1^1, \delta_1) \in$ IHS and is attached to ϕ_n **and** $\phi_1^1\delta_1$ is smaller than $\phi_n^1\theta_n$, where θ_n is the cumulative substitution $\sigma_n^1 \ldots \sigma_n^{m_n}$ **then**
      if $n == 1$ **then**
        { 1-*cycle is found !*}
        ok_IHs := ok_IHs $\cup$ $\{(\phi_1^1, \delta_1)\}$
        IHS := IHS $\setminus$ $\{(\phi_1^1, \delta_1)\}$
      else
        if for each $i \in [1..n-1]$: $(\phi_{i+1}^1, \delta_{i+1}) \in$ IHS and is attached to ϕ_i, and $\phi_{i+1}^1\delta_{i+1}$ is smaller than $\phi_i^1\theta_i$, where θ_i is the cumulative substitution $\sigma_i^1 \ldots \sigma_i^{m_i}$ **then**
          {*n-cycle ($n > 1$) is found !*}
          ok_IHs := ok_IHs $\cup \cup_{i=1}^n$ $\{(\phi_i^1, \delta_i)\}$
          IHS := IHS $\setminus$ $\cup_{i=1}^n$ $\{(\phi_i^1, \delta_i)\}$
        end if
      end if
    end if
  until no cycle is found
  return ok_IHs

Figure 2.1 illustrates a n-cycle. The relation between the conjecture attaching an IH and the IH is graphically represented with a non-horizontal arrow. The *cumulative* substitution θ_i written above the horizontal arrow pointing to a conjecture ϕ_i allows to produce it from $\phi_i^1\theta_i$ using a SPLIT-free derivation, for any $i \in [1..n]$. The main strength of the method is the fact that the induction reasoning involves only ordering constraints between instances of the conjectures starting the history chunks.

Implementation in SPIKE. The DRaCuLa strategy has been implemented in the SPIKE implicit induction prover. More details about its usage are given in Chapter 4.

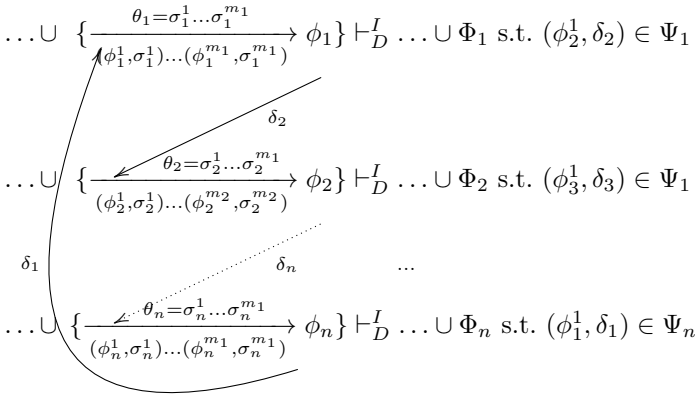


Figure 2.1: A cycle for checking n IHs.

2.3 Building cycles

The cycle identification process may be costly if there are considered all the permutations built from subsets of the stand-by conjectures from the current state. Since each permutation represents a potential cycle, all the ancestors of the conjectures from the permutation may be tested to satisfy the ordering constraints. A more efficient alternative that avoids this combinatorial explosion problem is to build cycles incrementally and by need, as follows. Anytime a new IH (ϕ, δ) attached to a conjecture ϕ_0 is about to be checked and ϕ is not yet proved, the strategy for choosing the current conjecture privileges the offsprings of ϕ from the current state that already participate in the cycle. If the corresponding ordering constraints are satisfied, the cycle is built. Otherwise, the ordering constraint related to the conjecture initiating the history chunk of ϕ_0 will be verified. If it is satisfied, a new history chunk starting with ϕ is added to the cycle. In this case, the proof of the offspring of ϕ continues to be developed until either it is proved, or a new IH required during the proof development is to be checked as previously. If the ordering constraint is not satisfied, INDUCTION cannot be applied on ϕ and the proof of ϕ continues to be developed either by checking other IHs or by applying a rule other than INDUCTION.

Definition 2 (D-proof) *Any D-derivation built by Develop(E^0) and finishing with an empty set of conjectures is a D-proof of E^0, for any set of conjectures E^0.*

Theorem 9 (soundness of D) *For any set of conjectures E^0, if there is a D-proof of E^0 then $\models_{\mathcal{M}} E^0$.*

Proof By contradiction, assume that there is $\phi^0 \in E^0$ such that $\not\models_{\mathcal{M}} \phi^0$. Since Develop($E^0$) builds a derivation that finishes with an empty set of conjectures, there is a last step in the derivation when a false conjecture, denoted by ϕ', was processed. The applied rule is neither DEDUCTION, nor SPLIT because another false conjecture would be in the next step. So INDUCTION has to be applied on ϕ'. The derivation should include at least one cycle checking IHs attached to false conjectures such that the new conjectures resulted from the application of the INDUCTION rules from the cycle are true. Otherwise, the derivation does not perform inductive reasoning. More exactly, it can be transformed into a hierarchy of deductive proofs of conjectures from the proof of E^0, where the IHs are lemmas resulted from previously (deductively) proved conjectures, as follows. All the proofs that did not use IHs are at the bottom of the hierarchy, so they are true. The next upper level in the hierarchy consists of all proofs using as lemmas the conjectures proved at the bottom level, so they are true, too. And so on, by stepping up in the hierarchy level by level, the current level proofs use as lemmas only conjectures proved at a lower level. The hierarchy is bounded since the proof of E^0 has a finite number of conjectures. It results that all conjectures from the proof of E^0, including ϕ^0, are true. Contradiction, since ϕ^0 is false.

A classical induction reasoning will be performed on the number of cycles checking IHs attached to false conjectures in the proof of E^0.

The base case: We assume that there is only one n-cycle checking IHs attached on false conjectures such that for each $i \in [1..n]$, ϕ_i is explicitly represented in the cycle as $\xrightarrow[(\phi_i^1,\sigma_i^1)\ldots(\phi_i^{m_i},\sigma_i^{m_i})]{\theta_i=\sigma_i^1\ldots\sigma_i^{m_i}} \phi_i$ and has attached the IH $(\phi^1_{(i+1) \bmod n}, \delta_{(i+1) \bmod n})$. Moreover, $\phi^1_{(i+1) \bmod n}\delta_{(i+1) \bmod n}$ is smaller than $\phi_i^1\theta_i$, for any $i \in [1..n]$.

Any false instance of ϕ^0 should lead to one of the conjectures from the n-cycle, otherwise an extra cycle including false conjectures should exist in the proof. More exactly, for any counterexample $\phi^0\tau^0$, there is a conjecture ϕ from the n-cycle whose history is of the form $\xrightarrow[(\phi^0,\sigma^0)(\phi^1,\sigma^1).....(\phi^p,\sigma^p)]{} \phi$ and $\phi^0\tau^0$ is an instance of $\phi^0\sigma^0$. Moreover, any conjecture instance $\phi^i\theta^i$ is false, where θ^i is the cumulative substitution between ϕ^i and ϕ, for all $i \in [1..p]$. Otherwise, an extra cycle including false conjectures should exist in the proof, which leads to a contradiction.

W.l.o.g, we assume that ϕ is represented in the n-cycle by ϕ_n and the INDUCTION rule applied on ϕ_n is $E_n \cup \{\phi_n\} \vdash_D^I E_n \cup \Phi_n$ using the non-empty set Ψ_n of IHs such that $\Phi_n \cup \Psi_n \models_{\mathcal{M}} \phi_n$ and $\not\models_{\mathcal{M}} \phi_n$. Φ_n is valid, otherwise there is an extra cycle including false conjectures which is applied in the proof of Φ_n. Since Φ_n is valid and $\Phi_n \cup \Psi_n \models_{\mathcal{M}} \phi_n$, there is a false IH in Ψ_n. We can show that the IHs from Ψ_n, different from (ϕ_1^1, δ_1), are true. More exactly, if (ψ, δ) is such an IH, it cannot be proved only

with deductive reasoning, so there should exist a new cycle checking ψ. This cycle cannot have false conjectures, so it includes neither ϕ_n nor any false offsprings of ψ. Therefore, ψ is true, which also holds for $\psi\delta$. We conclude that (ϕ_1^1, δ_1) is the only false IH attached to ϕ_n.

The history chunk of ϕ_n is represented in the n-cycle as $\xrightarrow[(\phi_n^1,\sigma_n^1)\ldots(\phi_n^{m_n},\sigma_n^{m_n})]{\theta_n=\sigma_n^1\ldots\sigma_n^{m_n}} \phi_n$. Since $\phi_n^1\theta_n$ is false, let $\phi_n^1\theta_n\tau$ be a minimal counterexample of it. Moreover, $\phi_n\tau$ is a counterexample because no cycle including false conjectures can be built on the path leading from $\phi_n^1\sigma_n^1$ to ϕ_n. Thanks to the 'stability under substitutions' property of the induction ordering, we deduce that $\phi_1^1\delta_1\tau$ is false and smaller than $\phi_n^1\theta_n\tau$. For similar reasons, it cannot be proved outside the n-cycle, so it is an instance of $\phi_1^1\theta_1$. Let this false instance be $\phi_1^1\theta_1\tau_1$ such that $\phi_1^1\theta_1\tau_1 \equiv \phi_1^1\delta_1\tau$. A similar reasoning is performed on $\phi_1^1\theta_1\tau_1$ as for $\phi_n^1\theta_n\tau$ to show that there is a false instance $\phi_2^1\theta_2\tau_2 \equiv \phi_2^1\delta_2\tau_1$ which is smaller than $\phi_1^1\theta_1\tau_1$. And so on, there is a false instance $\phi_n^1\theta_n\tau_n \equiv \phi_n^1\delta_n\tau_{n-1}$ which is smaller than $\phi_{n-1}^1\theta_{n-1}\tau_{n-1}$. By the transitivity of the ordering, we have that $\phi_n^1\theta_n\tau_n$ is smaller than $\phi_n^1\theta_n\tau$. Moreover, it is false, so contradiction with the minimality assumption of $\phi_n^1\theta_n\tau$.

The step case: We assume that Develop(E^0) has produced a proof having m (> 1) cycles including false conjectures. By induction hypothesis, any proof using a smaller number of cycles with false conjectures is sound.

We follow a reasoning similar to that employed for the base case. We will focus on the last generated cycle from the proof of E^0 having false conjectures. Assuming that it is built from n (> 1) conjectures, each ϕ_i with $i \in [1..n]$ is explicitly represented as $\xrightarrow[(\phi_i^1,\sigma_i^1)\ldots(\phi_i^{m_i},\sigma_i^{m_i})]{\theta_i=\sigma_i^1\ldots\sigma_i^{m_i}} \phi_i$ and has attached the IH $(\phi^1_{(i+1) \bmod n}, \delta_{(i+1) \bmod n})$. Moreover, $\phi^1_{(i+1) \bmod n}\delta_{(i+1) \bmod n}$ is smaller than $\phi_i^1\theta_i$, for any $i \in [1..n]$.

It can be noticed that any false instance of ϕ^0 should lead to one of the conjectures from the n-cycle, otherwise the proof of such a false instance should have 'less than m' cycles having false conjectures since the last cycle is not included in the proof. By the induction hypothesis, the proof is sound, so contradiction with the assumption that the instance of ϕ^0 is false. Therefore, for any counterexample $\phi^0\tau^0$ that is instance of $\phi^0\sigma^0$, there is a conjecture ϕ from the n-cycle whose history is of the form $\xrightarrow[(\phi^0,\sigma^0)(\phi^1,\sigma^1)\ldots\ldots(\phi^p,\sigma^p)]{} \phi$. In addition, any conjecture instance $\phi^i\theta^i$ is false, where θ^i is the cumulative substitution between ϕ^i and ϕ, for all $i \in [1..p]$. Otherwise, $\phi^0\tau^0$ can be proved with 'less than m' cycles having false conjectures, so contradiction.

Again, we assume that ϕ is represented in the n-cycle by ϕ_n and the INDUCTION rule applied on ϕ_n is $E_n \cup \{\phi_n\} \vdash_D^I E_n \cup \Phi_n$ using the non-empty set Ψ_n of IHs such that $\Phi_n \cup \Psi_n \models_{\mathcal{M}} \phi_n$ and $\not\models_{\mathcal{M}} \phi_n$. Φ_n is valid, otherwise there is an extra cycle including false conjectures which is applied in the proof of Φ_n which contradicts the assumption that the n-cycle is the last generated one in the proof of E^0. Since Φ_n is valid and $\Phi_n \cup \Psi_n \models_{\mathcal{M}} \phi_n$, there is a false IH in Ψ_n. The IHs from Ψ_n, excepting (ϕ_1^1, δ_1), are true. More exactly, if (ψ, δ) is such an IH, it has to be proved with 'less than m' cycles having false conjectures since the n-cycle cannot be included in the proof, i.e., either (ψ, δ) is checked by a cycle including ϕ_n and generated before the n-cycle, or ψ was proved before generating the n-cycle. For the last case, ψ is true by induction hypothesis, which also holds for $\psi\delta$. We conclude that (ϕ_1^1, δ_1) is the only false IH attached to ϕ_n.

As for the base case, we represent the history chunk of ϕ_n from the n-cycle as $\xrightarrow[(\phi_n^1,\sigma_n^1)\ldots(\phi_n^{m_n},\sigma_n^{m_n})]{\theta_n=\sigma_n^1\ldots\sigma_n^{m_n}} \phi_n$. Since $\phi_n^1\theta_n$ is false, we consider $\phi_n^1\theta_n\tau$ as being a minimal counterexample of it. In addition, $\phi_n\tau$ should be a counterexample, otherwise $\phi_n^1\theta_n\tau$ can be proved with 'less than m' cycles having false conjectures. Thanks to the 'stability under substitutions' property of the induction ordering, we deduce that $\phi_1^1\delta_1\tau$ is false and smaller than $\phi_n^1\theta_n\tau$. It should be an instance of $\phi_1^1\theta_1$, otherwise it can be proved outside the n-cycle using 'less than m' cycles including false conjectures. We denote this false instance by $\phi_1^1\theta_1\tau_1$, hence $\phi_1^1\theta_1\tau_1 \equiv \phi_1^1\delta_1\tau$. A similar reasoning is performed on $\phi_1^1\theta_1\tau_1$ as for $\phi_n^1\theta_n\tau$ to show that there is a false instance $\phi_2^1\theta_2\tau_2 \equiv \phi_2^1\delta_2\tau_1$ which is smaller than $\phi_1^1\theta_1\tau_1$. And so on, there is a false instance $\phi_n^1\theta_n\tau_n \equiv \phi_n^1\delta_n\tau_{n-1}$ which is smaller than $\phi_{n-1}^1\theta_{n-1}\tau_{n-1}$. By the transitivity of the ordering, we have that $\phi_n^1\theta_n\tau_n$ is smaller than $\phi_n^1\theta_n\tau$. Moreover, it is false, so contradiction with the minimality assumption

of $\phi_n^1\theta_n\tau$.

□

The DRaCuLa-based n-cycles from the D-proofs, for short *D-cycles* when n is not relevant, need less ordering constraints than the *reductive* cycles encountered in reductive induction derivations, for example the implicit induction and 'cyclic' proofs. More exactly, a reductive cycle requires that, for any history chunk $\xrightarrow[(\phi^0,\sigma^0)(\phi^1,\sigma^1)...(\phi^m,\sigma^m)]{} \phi$ with $m > 0$, $\phi^i\sigma^i$ to be greater than or (sometimes) equal to ϕ_{i+1}, for all $i \in [0..m-1]$. Moreover, $\phi^m\sigma^m$ should be greater than (or equal to) ϕ. As for the D-cycles, the IHs are instances of previous conjectures starting a history chunk, but they are required to be smaller than or equal to the conjecture they are applied to.

Lemma 3 *Any reductive n-cycle with an average of m conjectures per history chunk should satisfy $(m+1) \times n$ ordering constraints.*

Proof There are $n \times m$ ordering constraints concerning the conjectures from the history chunks, and n ordering constraints related to the IHs. □

For example, there is only one reductive cycle in the I_i^i-proof of $x + 0 = x$ from Example 6:

$$\xrightarrow[(x+0=x,\{x\mapsto S(x')\})]{\theta=\{x\mapsto S(x')\}} S(x' + 0) = S(x')$$
$$\xleftarrow{\delta=\{x\mapsto x'\}}$$

The two associated ordering constraints are: $S(x') + 0 = S(x')$ greater than $S(x' + 0) = S(x')$ which should be greater than $x' + 0 = x'$.

Lemma 4 *Any DRaCuLa-based n-cycle should satisfy n ordering constraints.*

Proof By the construction of D-cycles. □

One possible D-proof of $x + 0 = x$ can be built with the concrete inference system D_c:

DedNat (D_c): $E \cup \{\phi\} \vdash_{D_c} E \cup \Phi$,
  if either i) ϕ is a tautology; in this case Φ is empty,
    or ii) ϕ is rewritten by rewrite rules from Ax to ψ;
    in this case Φ is $\{\psi\}$.
SplitNat (S_c): $E \cup \{\phi[x]\} \vdash_{D_c} E \cup \{\phi[0], \phi[S(x')]\}$,
  where x' is a fresh natural variable.
IndNat (I_c): $E \cup \{\phi\} \vdash_{D_c} E \cup \Phi$,
  if ϕ is rewritten with an IH that is checked by a D-cycle.

Example 8 (D_c-proof of $x + 0 = x$) *The generated proof is: $\{\underline{x + 0 = x}\} \vdash_{D_c}^{S_c} \{\underline{0 + 0 = 0},\ S(x') + 0 = S(x')\} \vdash_{D_c}^{D_c(2)} \{\underline{S(x') + 0 = S(x')}\} \vdash_{D_c}^{D_c} \{\underline{S(x' + 0) = S(x')}\} \vdash_{D_c}^{I_c} \{\underline{S(x') = S(x')}\} \vdash_{D_c}^{D_c} \emptyset$. $\vdash_{D_c}^{D_c(2)}$ means that* DedNat *firstly rewrites with the axioms defining '+', then deletes the resulted tautology. The IH from the* IndNat *step is checked by the D-cycle:*

$$\xrightarrow[(x+0=x,\{x\mapsto S(x')\});(S(x')+0=S(x'),\{x'\mapsto x'\})]{\theta=\{x\mapsto S(x')\}} S(x' + 0) = S(x')$$
$$\xleftarrow{\delta=\{x\mapsto x'\}}$$

Even if the history chunk has more conjectures, there is only one ordering constraint to be satisfied, i.e., $x' + 0 = x'$ should be smaller than $S(x') + 0 = S(x')$.

DEDNAT (resp. SPLITNAT, resp. INDNAT) implements DEDUCTION (resp. SPLIT, resp. INDUCTION). Therefore D_c is sound since D is sound, by Theorem 9.

Lemma 5 *Any reductive n-cycle is a DRaCuLa-based n-cycle.*

Proof Assume that a reductive n-cycle exists, represented as a non-empty list of n conjectures $\phi_1, \ldots, \phi_n$ of the form $\xrightarrow[\ldots(\phi_i^1,\sigma_i^1)\ldots(\phi_i^{m_1},\sigma_i^{m_1})]{\theta_i=\sigma_i^1\ldots\sigma_i^{m_1}} \phi_i$, $i \in [1..n]$. By construction, for each $i \in [1..n]$, the reductive constraints require that $\phi_i^k\sigma_i^k$ be greater than (and sometimes equal to) ϕ_i^{k+1} ($k \in [1..m_1-1]$) and $\phi_i^{m_1}\sigma_i^{m_1}$ be greater than (or equal to) ϕ_i. By instantiating each $\phi_i^k\sigma_i^k$ by $\sigma_i^{k+1}\ldots\sigma_i^{m_1}$ and due to the 'stability under substitution' of the ordering, it results the decreasing sequence $\phi_i^1\theta_i^1$, $\phi_i^2\theta_i^2$, $\ldots$, $\phi_i^{m_1}\theta_i^{m_1}$, ϕ_i, where θ_i^j is the cumulative substitution $\sigma_i^j\ldots\sigma_i^{m_1}$. By the transitivity of the ordering relation, $\phi_i^1\theta_i^1$ is greater than ϕ_i. Moreover, by construction, ϕ_i should be greater than or equal to the IH $\phi^1_{(i+1)\bmod n}{}^{\delta_{(i+1)\bmod n}}$. Applying again the transitivity property of the ordering relation, $\phi_i^1\theta_i^1$ is greater than $\phi^1_{(i+1)\bmod n}{}^{\delta_{(i+1)\bmod n}}$.

Therefore, the reductive n-cycle is a DRaCuLa-based n-cycle since $\phi_i^1\theta_i^1$ is greater than $\phi^1_{(i+1)\bmod n}{}^{\delta_{(i+1)\bmod n}}$, forall $i \in [1..n]$. □

Theorem 10 (generalisation of reductive induction) *Any reductive induction proof is a D-proof.*

Proof Given a reductive induction proof, its reductive cycles can be represented as D-cycles, by Lemma 5. Moreover, the DRaCuLa strategy requires no ordering constraints for the proof parts outside the reductive cycles. □

The DRaCuLa-based proofs are more flexible in terms of induction orderings because the ordering constraints inside the D-cycles can be formulated with different induction orderings. This is not the case for the reductive proofs which are governed by only one global induction ordering that should satisfy all the reductive constraints. As a side-effect, the axioms and IHs involved in a proof may satisfy additional constraints. For example, the term ordering used by rewrite-based specifications to orient the axioms into rewrite rules should be *compatible* with the induction ordering, i.e., the union of the two ordering relations should be an induction ordering. The specifications fitting for DRaCuLa-based reasoning no longer need them, those adapted for term-based Noetherian induction reasoning [Walther, 1994] being good candidates.

Theorem 11 (generalisation of term-based Noetherian induction) *Any term-based Noetherian induction proof can be customized to a '1-cycle'-based D-proof.*

Proof According to the term-based Noetherian induction principle, one can use $\phi^1(\overline{k})$ as IH in the proof of $\phi^1(\overline{m})$, as long as $\overline{k} <_t \overline{m}$. On the other hand, according to Theorem 7, the ordering constraint can be reformulated using a formula-based Noetherian induction principle as: $\phi^1(\overline{k})$ should be smaller than $\phi^1(\overline{m})$, by considering the measure value of $\overline{v}$ as being that for $\phi^1(\overline{v})$, for any term vector $\overline{v}$. Assuming that $\phi^1(\overline{k})$ is an IH attached to the offspring ϕ_f of $\phi^1(\overline{m})$, the induction principle can be schematized by the 1-cycle

$$\xrightarrow[(\phi^1\sigma)\ldots(\phi',\sigma')]{\theta=\sigma\ldots\sigma'} \phi_f, \qquad \phi_f \xleftarrow{\ \delta\ } \phi^1(\overline{k})$$

where $\phi^1(\overline{m})$ is (ϕ^1, θ) and $\phi^1(\overline{k})$ is (ϕ^1, δ). □

Example 9 (customisation of a Peano induction proof) *The Peano induction proof of $x + 0 = x$ from Example 4 builds a 1-cycle similar to the D-cycle issued from the reductive I_i-proof from Example 6, excepting that the induction ordering is defined over terms. Its customisation to a D-cycle is done by defining the measure value of the equality $x + 0 = x$ as being the term x, for any natural x. Therefore, the constraint '$S(x') + 0 = S(x')$ greater than $x' + 0 = x'$' boils down to '$S(x')$ greater than x''.*

2.4 Managing mutual induction

The following set of axioms mutually defines over the naturals the functions *even*, *odd* and their conditional versions *even*1 and *odd*1, respectively:

$$even(0) = True \tag{2.1}$$
$$even(S(0)) = False \tag{2.2}$$
$$even(S(S(x))) = even1(S(S(x)) + 0) \tag{2.3}$$

$$even1(0) = True \tag{2.4}$$
$$even1(S(0)) = False \tag{2.5}$$
$$odd(x) = False \Rightarrow even1(S(S(x))) = even(x) \tag{2.6}$$
$$odd(x) = True \Rightarrow even1(S(S(x))) = False \tag{2.7}$$

$$odd(0) = False \tag{2.8}$$
$$odd(S(0)) = True \tag{2.9}$$
$$odd(S(S(x))) = odd1(S(S(x)) + 0) \tag{2.10}$$

$$odd1(0) = False \tag{2.11}$$
$$odd1(S(0)) = True \tag{2.12}$$
$$even(x) = True \Rightarrow odd1(S(S(x))) = odd(x) \tag{2.13}$$
$$even(x) = False \Rightarrow odd1(S(S(x))) = True \tag{2.14}$$

It can be noticed that all defined functions are terminating, sufficiently complete and consistent.

The conjectures to be proved are: $\phi_1^1 : even(x + x) = True$ and $\phi_2^1 : odd(y + y) = False$, using induction reasoning to be performed w.r.t. the initial model of the axioms. We assume that the previous proved conjecture $x + 0 = x$ is available as lemma, as well as the lemma $x + S(y) = S(x + y)$ which can be similarly proved. The ordering over conditional equalities is the multiset extension of the rpo ordering based on the precedence $<_F$ and equivalence $\sim_F$ relations over the function symbols: $True <_F False <_F 0 <_F S <_F + <_F (even \sim_F odd \sim_F even1 \sim_F odd1)$. The cyclic proof can be done using the inference system D_c if it is extended with the following rule:

DedCase (D_c'): $E \cup \{\phi\} \vdash_{D_c} E \cup \Phi$,
if a case analysis is performed on ϕ with rewrite rules
from Ax; in this case, Φ is made of the rewriting results.

The rewrite rules used by DedCase are conditional, of the form $d_1 = True \Rightarrow l_1 = r_1$ and $d_2 = False \Rightarrow l_2 = r_2$ such that both d_1 and d_2, resp. l_1 and l_2, are equal modulo renaming using the same renaming substitution. The case analysis is done as follows: if ϕ is an equality of the form $u = v$ such that u can be matched by l_1 and l_2 with the substitutions σ_1 and σ_2, respectively, then Φ consists of the set $\{d_1\sigma_1 = True \Rightarrow r_1\sigma_1 = v, d_2\sigma_2 = False \Rightarrow r_2\sigma_2 = v\}$. A tautology is an equality of the form $t = t$, or a conditional equality of the form $\ldots \Rightarrow t = t$ or $e \Rightarrow e$, for any term t and equality e.

The initial state of the D_c-proof is $\{\phi_1^1, \phi_2^1\}$. SplitNat is applied on ϕ_1^1 to result $\phi_1' : even(0 + 0) = True$ and $\phi_1^2 : even(S(x') + S(x')) = True$. ϕ_1' is rewritten by the axioms to the tautology $\phi_1 : True = True$, then deleted by DedNat. ϕ_1^2 is rewritten by the lemmas $x + S(y) = S(x + y)$ and $x + 0 = x$ to $\phi_1^3 : even1(S(S(x' + x'))) = True$. By case analysis with DedCase on ϕ_1^3, it results $\phi_1^{31} : odd(x' + x') = False \Rightarrow even(x' + x') = True$ and $\phi_1^{32} : odd(x' + x') = True \Rightarrow False = True$. IndNat is applied on ϕ_1^{31} by rewriting with the IH (ϕ_1^1, δ_1''), where $\delta_1'' = \{x \mapsto x'\}$. The IH is checked by

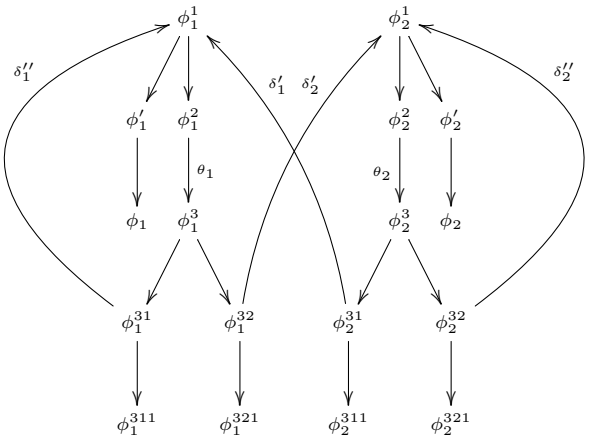


Figure 2.2: The skeleton of the D_c-proof.

the 1-cycle represented by $\xrightarrow[(\phi_1^1,\{x\mapsto S(x')\})(\phi_1^2,id)(\phi_1^3,id)]{\theta_1=\{x\mapsto S(x')\}} \phi_1^{31}$ since $\phi_1^1\delta_1''$ is smaller than $\phi_1^1\theta_1$. The rewriting operation produces the tautology $\phi_1^{311} : odd(x'+x') = True \Rightarrow True = True$ which is deleted by DedNat. Then, IndNat is expected to be applied on ϕ_1^{32} by rewriting this time with the IH (ϕ_2^1, δ_2'), where $\delta_2' = \{y \mapsto x'\}$. Hence, ϕ_1^{32} is put in stand-by and the proof of ϕ_2^1 starts by following similar steps as for ϕ_1^1. Firstly, SplitNat is applied to result $\phi_2' : odd(0+0) = False$ and $\phi_2^2 : odd(S(y')+S(y')) = False$. ϕ_2' is rewritten to the tautology $\phi_2 : False = False$, then deleted by DedNat. ϕ_2^2 is successively rewritten by the lemmas to $\phi_2^3 : odd1(y'+y') = False$ by DedNat, then by case analysis with DedCase to $\phi_2^{32} : even(y'+y') = True \Rightarrow odd(y'+y') = False$ and $\phi_2^{31} : even(y'+y') = False \Rightarrow True = False$. ϕ_2^{32} is further simplified to the tautology $\phi_2^{321} : even(y'+y') = True \Rightarrow False = False$ by IndNat with the IH (ϕ_2^1, δ_2''), where $\delta_2'' = \{y \mapsto y'\}$. The IH is checked by the 1-cycle with the history chunk $\xrightarrow[(\phi_2^1,\{y\mapsto S(y')\})(\phi_2^2,id)(\phi_2^3,id)]{\theta_2=\{y\mapsto S(y')\}} \phi_2^{32}$ since $\phi_2^1\delta_2''$ is smaller than $\phi_2^1\theta_2$. IndNat can also be applied on ϕ_2^{31} by rewriting with the IH (ϕ_1^1, δ_1'), where $\delta_1' = \{x \mapsto y'\}$. The 2-cycle consisting of the history chunks $\xrightarrow[(\phi_1^1,\{x\mapsto S(x')\})(\phi_1^2,id)(\phi_1^3,id)]{\theta_1=\{x\mapsto S(x')\}} \phi_1^{32}$ and $\xrightarrow[(\phi_2^1,\{y\mapsto S(y')\})(\phi_2^2,id)(\phi_2^3,id)]{\theta_2=\{y\mapsto S(y')\}} \phi_2^{31}$ can check the two IHs since $\phi_2^1\delta_2'$ is smaller than $\phi_1^1\theta_1$ and $\phi_1^1\delta_1'$ is smaller than $\phi_2^1\theta_2$. The two IndNat operations are further applied to give the tautologies $\phi_1^{321} : False = True \Rightarrow False = True$ and $\phi_2^{311} : True = False \Rightarrow True = False$, which are finally deleted by DedNat. The skeleton of the D_c-proof is given in Figure 2.2.

Let us notice that the conjectures ϕ_1^1 and ϕ_2^1 cannot be proved by reductive reasoning since the axioms cannot be simultaneously oriented from left to right and transformed into rewrite rules, in particular (2.3) and (2.6), as well as (2.10), (2.14) and the axioms defining '+'. The D-proof cannot either be redone by term-based Noetherian induction reasoning because of its 2-cycle.

Due to the unorientable axioms 2.3 and 2.10, SPIKE cannot produce an implicit induction proof of the conjectures. However, SPIKE succeeds to reproduce the cyclic proof using the DRaCuLa strategy. The cyclic proof script is available at http://code.google.com/p/spike-prover/.

2.5 Representing implicit induction proofs as cyclic proofs

Implicit induction proofs can be represented, as any reductive induction proofs, as D-proofs, according to Theorem 10. Table 2.1 gives some statistics about the implicit induction proofs of a bunch of conjectures involved in the validation process [Rusinowitch *et al.*, 2003] of a conformance algorithm for the ABR

#	conjecture	reductive constraints	IHs	cycles
1.	firstat_timeat	23	2	2
2.	firstat_progat	24	2	2
3.	sorted_sorted	5	0	0
4.	sorted_insat1	37	2	2
5.	sorted_insin2	45	2	2
6.	sorted_e_two	5	0	0
7.	member_t_insin	72	8	4
8.	member_t_insat	41	5	4
9.	member_firstat	37	3	3
10.	timel_insat_t	10	1	1
11.	erl_insin	11	1	1
12.	erl_insat	10	1	1
13.	erl_prog	38	2	2
14.	time_progat_er	20	1	1
15.	timeat_tcrt	16	1	1
16.	timel_timeat_max	43	1	1
17.	null_listat	17	2	2
18.	null_listat1	3	0	0
19.	cons_insat	4	1	1
20.	cons_listat	3	0	0
21.	progat_timel_erl	48	1	1
22.	progat_insat	156	4	2
23.	progat_insat1	63	3	2
24.	timel_listupto	7	0	0
25.	sorted_listupto	49	3	3
26.	time_listat	27	1	1
27.	sorted_cons_listat	62	2	2
28.	null_wind2	7	0	0
29.	timel_insin1	17	1	1
30.	null_listupto1	3	0	0
31.	erl_cons	11	0	0
32.	no_time	35	2	2
33.	final	29	2	2
	Total	978	54	46

Table 2.1: Statistics of the induction reasoning w.r.t. the ABR implicit induction proofs.

protocol [Rabadan and Klay, 1997] using SPIKE.[10] It illustrates the name of the conjectures, the number of reductive ordering constraints, the number of IHs and the number of D-cycles. The number of reductive ordering constraints is given by the number of reductive steps in the proof. For example, the implicit induction proof of the conjecture progat_insat requires 152 reductive ordering constraints but only 4 applications of IHs.

Interpreting implicit induction proofs as cyclic proofs allows to improve their certification process by Coq, as explained in Chapter 6, where every single proof step should be checked. On the one hand, the number of reductive ordering constraints, as indicated by Lemma 3, can be important. In [Stratulat, 2010], it has been shown that the validation of the ordering constraints for some proofs can last four times longer than for the validation of the deductive reasoning. On the other hand, the validation time can be dramatically reduced if the implicit induction proofs are interpreted as D-proofs because of the small number of non-reductive ordering constraints, as shown by Lemma 4. For example,

[10]The implicit induction and cyclic proof scripts and can also be accessed from http://code.google.com/p/spike-prover/

by inspecting the representation of the proof script of the conjecture `progat_insat` as a D-proof, it has been noticed that only two IHs are checked by 1-cycles, while the other two IHs do not need induction reasoning to be proved. Therefore, the validation process of the D-proof would require to check only 2 non-reductive ordering constraints. The IHs not requiring induction reasoning can be proved in priority using a different proof strategy, then considered as lemmas during the rest of the proof.

2.6 Conclusions

We have presented an induction proof technique that captures the first-order induction reasoning by the means of non-reductive cycles which can be built using different formula-based Noetherian induction orderings. It has been shown enough powerful to subsume any term-based and reductive formula-based Noetherian inductive proof methods by combining the best features of conventional and implicit induction proof techniques.

Implemented in SPIKE, the proposed technique can substantially diminish the number of ordering constraints required when certifying implicit induction proofs.

3

Building Term-based Noetherian Induction Proofs for Conjectures Proved with Formula-based Noetherian Induction Principles

Sommaire

3.1 Coq experiments 26

3.1.1 Proving by cyclic induction 27

3.1.2 Proving by structural induction 29

3.2 Lazy generation of induction schemas for algorithm synthesis 36

3.2.1 Sorting of binary trees 37

3.2.2 Synthesis problem and method 38

3.2.3 Induction principles and algorithm extraction 38

3.2.4 Refining induction by lazy reasoning 39

3.3 Conclusions 41

This chapter presents different applications of Theorem 8, defining the conditions for which formula-based Noetherian induction principles can be *directly* converted to term-based Noetherian induction principles. We show that this conversion result can be extended at the proof level. Indeed, we give examples where structural induction proofs can be directly built from cyclic induction proofs. On the other hand, we show that the use of the explicit induction schemas, based on the variable instantiation schemas used in cyclic induction proofs, requires user creativity embodied in the definition of new lemmas and for guiding the way the explicit induction schemas are applied.

Structure of the chapter. This chapter has three sections. Section 3.1 reproduces Coq experiments published in [Stratulat, 2016] that aim at building term-based Noetherian induction proofs for conjectures already proved using formula-based Noetherian induction. In Section 3.2, we use the generation of explicit induction principles from formula-based Noetherian induction principles that are lazily built for synthesising sorting algorithms for binary trees. The results have been partially published in [Dramnesc *et al.*, 2019, Dramnesc *et al.*, 2016a, Dramnesc *et al.*, 2016b, Dramnesc *et al.*, 2015a, Dramnesc *et al.*, 2015b, Dramnesc *et al.*, 2015c]. The last section concludes.

3.1 Coq experiments

In this section, we firstly define a set of conjectures that can be proved by using cyclic induction, based on the variable instantiation schema of the Peano principle. All cyclic proofs follow a similar scenario. Next, we implement the cyclic induction reasoning in the Coq proof assistant [The Coq development team, 2020]. Finally, we show that the scenarios for proving these conjectures with Peano induction differ in terms of the number of induction steps and lemmas, as well as proof scenario. We identify three conjectures from this set that are hard or impossible to be proved by Peano induction.[11]

Related works. Previous works have tempted to convert term- to formula-based Noetherian induction proofs. Musser [Musser, 1980] was the first to compare conventional with inductionless induction methods. Since then, a lot of effort was put into clarifying the relations between explicit and implicit induction principles, [Garland and Guttag, 1988, Jouannaud and Kounalis, 1989, Kapur and Zhang, 1994, Naidich, 1996, Comon, 2001, Wirth, 2005] being among the most notable. Other studies have been conducted to reduce the gap between them. Protzen [Protzen, 1994] proposed a proof strategy to perform lazy induction on particular explicit induction proofs. Kapur and Subramaniam [Kapur and Subramaniam, 1996] devised a method that extends schemata-based induction to deal with a special class of mutually defined functions. Courant [Courant, 1996] identified a class of implicit induction inference systems for which the proofs can be reconstructed into conventional induction proofs. Reddy [Reddy, 1990] designed implicit induction inference rules that look similar to schemata-based induction rules. This feature has been implemented by subsequent formula-based induction systems [Bronsard and Reddy, 1991, Bouhoula *et al.*, 1992, Avenhaus *et al.*, 2003] to generate more compact and readable proofs. Instantiation results similar to ours have been achieved more recently for particular cases of *cyclic* proof systems. Sprenger and Dam [Sprenger and Dam, 2003] have shown the equivalence of two Gentzen-style proof systems for first-order μ-calculus with explicit approximations; one of them integrates local term-based induction rules, while the other lacks such induction rules. In turn, the second can build finite ω-regular proof trees for which the induction reasoning is argued by an external global induction discharge condition associated to the proof structure. In the same line, Brotherston and Simpson [Brotherston and Simpson, 2011] compared two classical first-order sequent calculus proof systems; the local induction is performed using conventional induction together with a rule that deals with a class of mutual inductive definitions. In this context, they showed that any proof using local induction arguments can be represented as a proof using global induction arguments and conjectured that the other direction also holds. Later, Berardi and Tatsuta proved that the conjecture does not hold, by providing the 2-Hydra example [Berardi and Tatsuta, 2019].

Defining the set of conjectures. Let $\mathcal{R}$ be a set of ternary inductive predicate symbols taking natural numbers as arguments such that each symbol $R \in \mathcal{R}$ is defined by a set of axioms of the form:

$$R(0, u, 0) \tag{3.1}$$
$$R(x_1, y_1, y_2) \Rightarrow R(S(x), u, 0) \tag{3.2}$$
$$R(x'_1, y'_1, y'_2) \Rightarrow R(0, u, S(v)) \tag{3.3}$$
$$R(x_1, y_1, y_2) \wedge R(x'_1, y'_1, y'_2) \Rightarrow R(S(x), u, S(v)) \tag{3.4}$$

where S is the 'successor' function and the variables x, u, v are universally quantified. The values of the parameters x_1, y_1, y_2 and x'_1, y'_1, y'_2 of R occurring in the condition part of the axioms are defined in order to satisfy the following ordering constraints: i) $R(x_1, y_1, y_2) < R(S(x), u, 0)$, ii) $R(x'_1, y'_1, y'_2) < R(0, u, S(v))$, iii) $R(x_1, y_1, y_2) < R(S(x), u, S(v))$, and iv) $R(x'_1, y'_1, y'_2) < R(S(x), u, S(v))$, by using a well-founded ordering $<$. This ordering is defined such that $R(z_1, z_2, z_3) < R(z'_1, z'_2, z'_3)$ if $\{\{z_1, z_1\}, \{z_2, z_2, z_3\}\} <$ $< \{\{z'_1, z'_1\}, \{z'_2, z'_2, z'_3\}\}$, for any naturals $z_1, z_2, z_3, z'_1, z'_2, z'_3$. Here, $\ll$ is the multiset extension of an ordering over multisets of terms which, in turn, is the multiset extension of the mpo ordering, defined over naturals and denoted by $<_t$, based on the precedence over the function symbols stating that 0 is

[11] The full Coq scripts of the proofs can be found at https://drive.google.com/file/d/19-8D4Cee8I4I_98RygwHYUtS-BbZrNMt/view?usp=sharing

smaller than S. It can be shown that this mpo ordering is well-founded and satisfies, for example, that $0 <_t S(x)$ and $x <_t S(x)$, for any natural x. Since every multiset extension of a well-founded ordering is also well-founded, we conclude that $<$ is well-founded.

We recall that a multiset A is smaller than another multiset B w.r.t. the *multiset extension* of some ordering $\prec$ if, after pairwisely deleting the common elements from A and B we get the multisets A' and B', respectively. In addition, for each element x in A', there is an element y in B' such that $x \prec y$. In our case, the ordering constraints

1. i) and iii) are, respectively, $\{\{x_1, x_1\}, \{y_1, y_1, y_2\}\} \ll \{\{S(x), S(x)\}, \{u, u, 0\}\}$ and $\{\{x_1, x_1\}, \{y_1, y_1, y_2\}\} \ll \{\{S(x), S(x)\}, \{u, u, S(v)\}\}$.

   They are satisfied if x_1 is an element from $\{0, u, x\}$ and the pair (y_1, y_2) an element from $\{(0,0), (0,x), (x,0), (0,S(x)), (x,S(x)), (x,x)\}$. Notice that the pairs of the form $(S(x), _)$ and the pairs including u cannot be assigned to (y_1, y_2);

2. ii) and iv) are, respectively, $\{\{x'_1, x'_1\}, \{y'_1, y'_1, y'_2\}\} \ll \{\{0, 0\}, \{u, u, S(v)\}\}$ and $\{\{x'_1, x'_1\}, \{y'_1, y'_1, y'_2\}\} \ll \{\{S(x), S(x)\}, \{u, u, S(v)\}\}$. They are also satisfied if x'_1 is an element from $\{0, u, v\}$ and (y'_1, y'_2) is from $\{(0,0), (0,u), (0,v), (u,v), (v,u), (u,0), (v,0), (v,v)\}$. The pair (u, u) cannot be assigned to (y'_1, y'_2).

Therefore, the set $\mathcal{R}$ will have $3 \times 6 \times 3 \times 8 = 432$ inductive predicate symbols. Finally, the set of conjectures for our purpose is $\{\forall x\ u\ v, R(x, u, v) \mid R \in \mathcal{R}\}$.

3.1.1 Proving by cyclic induction

For any $R \in \mathcal{R}$, the conjecture $\forall x\ u\ v, R(x, u, v)$ can be proved by cyclic induction reasoning using only variable instantiations and unfoldings with the axioms defining R, as shown in the proof digraph from Figure 3.1.

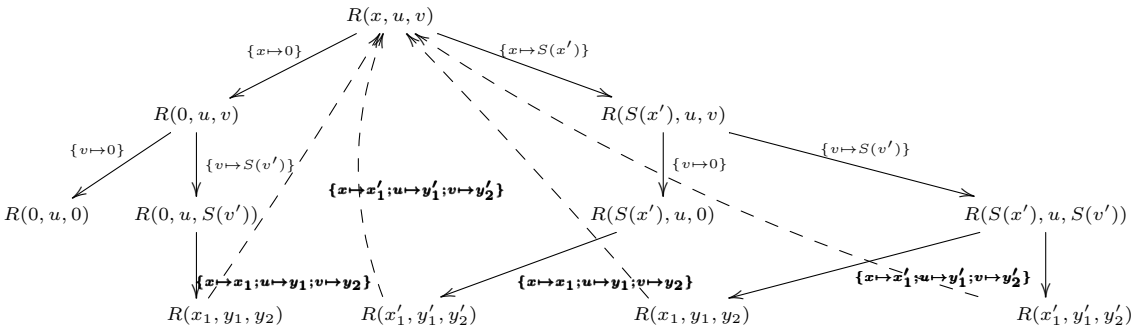


Figure 3.1: The digraph of the cyclic proof of $\forall x\ u\ v, R(x, u, v)$, for any $R \in \mathcal{R}$.

The root node is labeled by $R(x, u, v)$, the other nodes being labeled by inductive atoms that are instances of it. Each non root-node n is pointed by a solid arrow starting from some other node n'. If p' is the inductive atom labelling n', then the inductive atom labelling n results either i) by instantiating some variable from p' by 0 and $S(x)$, where x is a fresh variable, or ii) by unfolding p' using one of the conditional axioms (3.3)-(3.4). In the first case, the instantiating substitution annotates the corresponding solid arrow. The inductive atom labeling each leaf node either instantiates (3.1) or the inductive atom labeling the root node. In the last case, a dashed arrow is firstly created by leading the leaf node to the root node, then annotated with the instantiating substitution, written in boldface.

The proof digraph from Figure 3.1 contains cycles by following the arrows in the digraph. In general, not all proof derivations, for which the root formula is instantiated by leaf formulas, are sound. In our case, the soundness is guaranteed by the ordering constraints i) - iv), as shown by the cyclic induction method from [Stratulat, 2012], also presented in Chapter 2.

The Coq implementation. Let us assume that R is one of the inductive predicates symbols from $\mathcal{R}$, defined by the axioms:

$$R(0, u, 0) \tag{3.5}$$
$$R(u, x, S(x)) \Rightarrow R(S(x), u, 0) \tag{3.6}$$
$$R(v, u, v) \Rightarrow R(0, u, S(v)) \tag{3.7}$$
$$R(u, x, S(x)) \wedge R(v, u, v) \Rightarrow R(S(x), u, S(v)) \tag{3.8}$$

We will show how the cyclic induction reasoning for proving $\forall x\ u\ v, R(x, u, v)$ can be certified in Coq.

R can be specified in Coq as an inductive predicate, denoted here by **R**:

```
Inductive R: nat → nat → nat → Prop :=
r_1: ∀ u, R 0 u 0 |
r_2: ∀ x u, R u x (S x) → R (S x) u 0 |
r_3: ∀ u v, R v u v → R 0 u (S v) |
r_4: ∀ x u v, R u x (S x) → R v u v → R (S x) u (S v).
```

The scenario from the cyclic proof from Figure 3.1 can be reproduced if the conjecture to be proved is (temporarily) considered as an hypothesis before its usage.

```
Hypothesis R_admitted: ∀ x u v, R x u v.

Theorem R_assumption: ∀ x u v, R x u v.
Proof.
  destruct x ; intros.
  - Case "x=0". destruct v.
     + SCase "v=0". apply r_1.
     + SCase "v=S v". apply r_3. apply R_admitted.
  - Case "x=S x". destruct v.
    + SCase "v=0". apply r_2. apply R_admitted.
    + SCase "v=S v". apply r_4; apply R_admitted.
Qed.
```

The tactic `destruct`, when applied on a natural variable v, instantiates it by 0 and (S v).

It can be easily noticed that the cycles from the proof digraph from Figure 3.1 form a unique *strongly connected component*, i.e., a maximal sub-graph such that, given any two different nodes in it, there is a path between them in each direction. The induction reasoning performed along these cycles can be captured by an explicit induction schema issued from the definition of a terminating and recursive boolean function, denoted by f_P, taking as argument a triplet of naturals.

```
  Function f_P (a: nat × nat × nat) {wf (fun u v: nat× nat×nat ⇒ match u,v with
     (u1, x1, y1), (u2, x2, y2) ⇒ mless ({{[u1, u1]}} + {{[x1, x1, y1]}}) ({{[u2, u2]}}
+ {{[x2, x2, y2]}}) end) a}: bool :=
 match a with
     | (x', y) ⇒ match x' with
                 | (u, x) ⇒
                   match u, x, y with
                     | 0, _, 0 ⇒ true
                     | (S x), u, 0 ⇒ f_P (u, x, (S x))
                     | 0, u, (S v) ⇒ f_P (v, u, v)
                     | (S x), u, (S v) ⇒ andb (f_P (v, u, v)) (f_P (u, x, (S x)))
                   end
                end
```

```
end.
```

The function f_P firstly decomposes the triplet given as argument, then performs a case analysis on the resulting naturals to finally get a different (functional) representation of the definition of **R**.

As any function whose definition is accepted by Coq, f_P should terminate. The Coq environment generates proof obligations requiring that its argument should decrease after each recursive call w.r.t. the well-founded ordering provided after the `wf` keyword, where mless is the multiset extension of the multiset extension of the 'less than' syntactic ordering over naturals, defined using the CoLoR library [Blanqui and Koprowski, 2011].[12] Once the proof obligations are discharged, Coq automatically generates a functional (induction) schema f_P_ind.

An explicit induction proof can be built for the theorem RP_true which is similar to R_assumption when using RP, the version of **R** with only one (triplet) argument.

```
Definition RP z := R (fst (fst z)) (snd (fst z)) (snd z).

Theorem RP_true: ∀ u x y, RP (u, x, y).
Proof.
  intros u x y. pattern (u, x, y). pattern (f_P (u, x, y)).
  apply f_P_ind; intros; unfold RP; simpl. (* apply the induction schema *)
  apply r_1.                                (* follow the cyclic proof *)
  apply r_2; unfold RP in H; simpl in H; trivial.
  apply r_3. unfold RP in H; simpl in H. trivial.
  apply r_4; unfold RP in H0; simpl in H0; trivial.
Qed.
```

Finally, the *R_admitted* hypothesis can be proved.

```
Theorem R_admitted: ∀ x u v, R x u v.
  (* the proof follows directly from RP_true *)
```

Similarly, $\forall\ x\ u\ v$, $R'(x,u,v)$ can be certified in Coq, for any $R' \in \mathcal{R}$, using a similar scenario for which only the termination proof is different.

3.1.2 Proving by structural induction

The cyclic proofs of the conjectures $\forall\ x\ u\ v$, $R'(x,u,v)$, where $R' \in \mathcal{R}$ only instantiate natural variables using 0 and the successor of new variables. The explicit induction schema that is based on this variable instantiation schema is the *Peano induction*, a structural induction principle issued from the analysis of the recursive definition of naturals. To recall it, in order to prove a formula $P(x)$, where x is a natural variable to be instantiated, also called *induction variable*, it is enough to prove both $P(0)$ and $\forall x', P(x') \Rightarrow P(S(x'))$, where $P(x')$ is an *induction hypothesis* and x' a fresh variable. $P(x')$ can be soundly used in the proof of $P(S(x'))$ because the number of 'S' symbols in x' is smaller than in $S(x')$, for any natural x', and the 'less than' ordering over naturals is well-founded. In Coq, the Peano induction is automatically generated from the recursive definition of the **nat** datatype. It can be applied using the tactic `induction` which takes as argument the induction variable.

To distinguish each R' from $\mathcal{R}$, they are represented under the form of $Rijkl$, where

- i is the position of x_1 from the axioms (3.2) and (3.4) in the list $[0, u, x]$,
- j is the position of (y_1, y_2) from the axioms (3.2) and (3.4) in the list $[(0,0),(0,x),(x,0),(0,S(x)),(x,S(x)),(x,x)]$,
- k is the position of x'_1 from the axioms (3.3) and (3.4) in the list $[0, u, v]$, and
- l is the position of (y'_1, y'_2) from (3.3) and (3.4) in $[(0,0),(0,u),(0,v),(u,v),(v,u),(u,0),(v,0),(v,v)]$.

[12] More details about the definition of syntactic orderings in Coq is given in Chapter 6.

For example, the symbol R, defined by the axioms (3.5)-(3.8), will be referred to as $R2534$.

Similarly in Coq, the axioms defining $Rijkl$, denoted by **R**$ijkl$, are labelled as r$ijkl$_1, r$ijkl$_2, r$ijkl$_3, and r$ijkl$_4. In addition, the theorem to be proved is denoted as R$ijkl$_true.

All proofs of R$ijkl$_true have the following structure:

```
Theorem Rijkl_true: ∀ x u v, Rijkl x u v.
Proof.
  destruct x; destruct v; intros.
  (* case x=0, v=0 *) apply rijkl_1.
  (* case x=0, v=(S v) *) apply rijkl_3. (* to complete *)
  (* case x=(S x), v=0 *) apply rijkl_2. (* to complete *)
  (* case x=(S x), v=(S v) *) apply rijkl_4. (*to complete*)
Qed.
```

This proof scenario is similar to that from Figure 3.1, excepting that the proof of the formulas labelling the lowest nodes in the proof digraph should be provided.

Different proof scenarios can be distinguished to complete the proof of $Rijkl$_true. Table 3.1 presents some proof statistics for each case, where the second (resp., third) column gives the number of `induction` (resp., `destruct`) calls, and the fourth column the number of times '`apply` r$ijkl$_4' was called. The fifth column displays the number of intermediate lemmas and the last column the depth of the proof tree, issued by expanding the calls to the lemmas and by considering as one 'big step' each sequence of steps different from `induction`, `destruct`, `split` (for dealing with conjunctions) and '`apply` r$ijkl$_4'. We do not claim that our proofs have a digraph with minimal depth, our intention is only to give an idea about the degree of difficulty of each proof.

Scenario 1: no induction steps. This is the most trivial one. We consider the case **R1111**.

```
Theorem R1111_true: ∀ x u v, R1111 x u v.
Proof.
  destruct x; destruct v; intros.
  apply r1111_1.
  apply r1111_3. apply r1111_1.
  apply r1111_2. apply r1111_1.
  apply r1111_4; apply r1111_1.
Qed.
```

Overall, 78 cases have no induction steps in their proofs. These proofs are performed mainly by instantiating variables and unfolding axioms, the maximal depth being of 7 (e.g., for R1132_true).

Scenario 2: one induction step. We consider the case **R1113** where an induction step is performed in the proof of the additional lemma.

```
Theorem R1113_10v: ∀ v, R1113 0 0 v.
Proof.
  induction v.
   - Case "v=0" . apply r1113_1.
   - Case "v=S v". apply r1113_3. apply IHv.
Qed.
```

```
Theorem R1113_true: ∀ x u v, R1113 x u v.
Proof.
  destruct x; destruct v; intros. apply r1113_1.
  apply r1113_3. apply R1113_10v.
  apply r1113_2. apply r1113_1.
  apply r1113_4; apply R1113_10v.
Qed.
```

Scenario 3: two induction steps. We detail the case **R1115** where two induction steps are performed in the proof of the conjunction lemma:

```
Theorem R1115_mix: ∀ u v, R1115 0 v u ∧ R1115 0 u v.
Proof.
  induction u; intros.
  - Case "u=0". split. apply r1115_1. destruct v. apply r1115_1. apply r1115_3. apply
r1115_1.
  - Case "u=S u". split. apply r1115_3. apply IHu. induction v.
    + SCase "v=0". apply r1115_1.
    + SCase "v = S v". apply r1115_3. apply r1115_3. apply IHu.
Qed.

Theorem R1115_true: ∀ x u v, R1115 x u v.
Proof.
  destruct x; destruct v; intros. apply r1115_1.
  apply r1115_3. apply R1115_mix.
  apply r1115_2. apply r1115_1.
  apply r1115_4; apply R1115_mix.
Qed.
```

Scenario 4: more than two induction steps. We chose to comment the cases **R2535** and **R2534**. The proof of R2535_true required 3 induction steps and its digraph has the deepest depth (19). On the other hand, in spite of our efforts, the proof of R2534_true, as well as R2634_true and R2636_true, could not have been completed.

Statistics for the Coq proofs.

Table 3.1: Statistics about the proofs by Peano induction of *Rijkl*_true.

Case	IS	DS	CR	Lemmas	Depth	Case	IS	DS	CR	Lemmas	Depth
R1111	0	2	1	0	3	**R2411**	0	3	2	1	5
R1112	0	3	1	1	4	**R2412**	0	4	2	2	5
R1113	1	2	1	1	4	**R2413**	1	3	2	2	6
R1114	1	2	1	1	4	**R2414**	1	3	2	2	6
R1115	2	2	1	1	6	**R2415**	1	5	2	2	5
R1116	0	2	1	0	3	**R2416**	0	3	2	1	5
R1117	0	2	1	0	3	**R2417**	0	3	2	1	5
R1118	1	2	1	1	4	**R2418**	1	3	2	2	6
R1121	0	3	1	1	4	**R2421**	0	4	2	2	6
R1122	1	2	2	1	5	**R2422**	0	4	2	2	5
R1123	1	4	2	1	6	**R2423**	3	6	3	2	9
R1124	1	4	2	1	6	**R2424**	4	5	3	2	9
R1125	2	2	3	1	7	**R2425**	1	4	5	2	9
R1126	0	3	1	1	4	**R2426**	0	4	2	2	6
R1127	0	3	1	1	4	**R2427**	0	4	2	2	6
R1128	1	4	2	1	6	**R2428**	1	5	5	3	9
R1131	0	3	1	1	4	**R2431**	1	3	2	2	6
R1132	0	6	2	1	7	**R2432**	4	8	3	2	9
R1133	1	2	2	1	5	**R2433**	1	3	3	2	7
R1134	1	2	2	1	5	**R2434**	7	5	11	2	12
R1135	1	6	3	1	8	**R2435**	3	6	7	2	12
R1136	0	4	1	1	5	**R2436**	3	5	2	2	7
R1137	0	3	1	1	4	**R2437**	1	6	4	2	8

Table 3.1 – Continued

Case	IS	DS	CR	Lemmas	Depth	Case	IS	DS	CR	Lemmas	Depth
R1138	1	2	2	1	5	**R2438**	1	3	4	2	8
R1211	0	3	1	1	4	**R2511**	1	3	4	1	7
R1212	0	3	1	1	4	**R2512**	1	5	4	2	8
R1213	1	2	1	1	3	**R2513**	2	4	4	2	8
R1214	1	2	1	1	4	**R2514**	2	5	3	2	9
R1215	1	4	1	1	5	**R2515**	2	6	4	2	9
R1216	0	3	1	1	4	**R2516**	1	4	4	2	8
R1217	0	3	1	1	4	**R2517**	1	4	4	1	7
R1218	1	3	1	2	4	**R2518**	2	4	4	2	8
R1221	0	4	1	2	4	**R2521**	2	4	4	2	8
R1222	0	4	2	2	6	**R2522**	2	4	5	2	9
R1223	1	4	2	2	7	**R2523**	4	6	4	2	11
R1224	2	4	2	2	6	**R2524**	4	4	7	2	11
R1225	1	4	3	2	8	**R2525**	3	6	16	3	16
R1226	0	4	1	2	5	**R2526**	2	4	8	2	11
R1227	0	4	1	2	5	**R2527**	2	5	8	3	11
R1228	1	5	3	3	10	**R2528**	2	6	17	3	18
R1231	1	2	1	3	5	**R2531**	2	4	4	2	8
R1232	1	4	2	3	8	**R2532**	2	6	7	3	13
R1233	1	3	2	3	7	**R2533**	2	4	7	2	9
R1234	2	2	3	3	8	**R2534**	-	-	-	-	-
R1235	2	5	3	3	9	**R2535**	3	7	18	3	19
R1236	1	3	1	3	6	**R2536**	2	5	9	3	13
R1237	1	2	1	3	5	**R2537**	2	4	11	2	12
R1238	1	2	1	3	6	**R2538**	2	4	14	2	13
R1311	0	2	1	0	3	**R2611**	1	5	3	1	8
R1312	0	3	1	1	4	**R2612**	1	8	3	2	8
R1313	1	2	1	1	4	**R2613**	2	7	3	2	9
R1314	1	2	1	1	4	**R2614**	2	8	4	2	11
R1315	1	4	1	1	5	**R2615**	2	5	4	2	10
R1316	0	3	1	1	4	**R2616**	1	6	3	1	8
R1317	0	2	1	0	3	**R2617**	1	6	3	1	8
R1318	1	2	1	1	4	**R2618**	2	7	3	2	9
R1321	0	3	1	1	4	**R2621**	2	8	3	2	10
R1322	0	3	1	1	4	**R2622**	2	8	5	2	12
R1323	1	4	2	1	6	**R2623**	6	9	5	3	14
R1324	1	4	2	1	6	**R2624**	10	2	7	3	13
R1325	1	3	3	1	8	**R2625**	3	7	17	3	18
R1326	0	3	1	1	4	**R2626**	2	6	5	2	14
R1327	0	3	1	1	4	**R2627**	2	7	6	3	15
R1328	1	4	2	1	6	**R2628**	2	7	8	2	16
R1331	0	3	1	1	4	**R2631**	2	7	3	2	12
R1332	0	7	2	1	7	**R2632**	4	10	5	3	13
R1333	1	2	2	1	5	**R2633**	2	6	4	2	13
R1334	1	2	2	1	5	**R2634**	-	-	-	-	-
R1335	1	6	3	1	8	**R2635**	3	9	11	4	16
R1336	0	3	1	1	4	**R2636**	-	-	-	-	-
R1337	0	3	1	1	4	**R2637**	2	7	6	2	15
R1338	1	2	2	1	5	**R2638**	2	7	6	2	14

Table 3.1 – Continued

Case	IS	DS	CR	Lemmas	Depth	Case	IS	DS	CR	Lemmas	Depth
R1411	0	2	1	1	3	**R3111**	1	2	1	1	4
R1412	0	3	1	2	4	**R3112**	1	3	1	1	4
R1413	1	2	1	2	4	**R3113**	2	2	1	2	4
R1414	1	2	1	2	4	**R3114**	2	2	1	2	4
R1415	1	4	1	1	5	**R3115**	2	4	1	2	5
R1416	0	3	1	2	4	**R3116**	1	2	1	1	4
R1417	0	2	1	1	3	**R3117**	1	2	1	1	4
R1418	1	2	1	2	4	**R3118**	2	2	1	2	4
R1421	0	3	1	2	4	**R3111**	1	2	1	1	4
R1422	0	3	2	2	5	**R3122**	1	3	2	2	5
R1423	3	4	2	1	7	**R3123**	4	4	2	2	7
R1424	2	4	2	2	6	**R3124**	2	4	2	2	7
R1425	1	3	3	2	7	**R3125**	2	4	4	2	8
R1426	0	3	1	2	4	**R3126**	1	3	1	2	4
R1427	0	3	1	2	4	**R3127**	1	3	1	2	5
R1428	2	4	3	3	8	**R3128**	3	5	4	2	9
R1431	1	2	1	2	4	**R3131**	1	3	1	2	5
R1432	4	5	2	1	8	**R3132**	1	4	2	2	7
R1433	1	2	2	2	5	**R3133**	2	2	2	2	6
R1434	2	2	3	2	5	**R3134**	2	2	2	2	6
R1435	2	5	3	3	9	**R3135**	4	8	4	2	11
R1436	1	3	1	3	5	**R3136**	1	4	1	2	6
R1437	1	2	1	2	4	**R3137**	1	3	1	2	5
R1438	1	2	2	2	5	**R3138**	2	2	2	2	6
R1511	0	2	1	1	3	**R3211**	1	2	2	1	5
R1512	0	3	1	2	4	**R3212**	1	3	2	2	5
R1513	1	2	1	1	4	**R3213**	1	3	2	2	5
R1514	1	3	1	2	5	**R3214**	3	2	2	2	6
R1515	1	4	1	2	5	**R3215**	2	4	2	2	5
R1516	0	3	1	2	4	**R3216**	1	2	2	1	5
R1517	0	2	1	1	3	**R3217**	1	2	2	1	5
R1518	1	2	1	2	4	**R3218**	2	2	2	2	6
R1521	1	2	1	2	4	**R3211**	1	3	2	2	6
R1522	1	2	2	2	5	**R3222**	2	2	3	2	5
R1523	4	4	3	3	7	**R3223**	5	4	3	2	7
R1524	3	2	2	2	6	**R3224**	3	4	3	2	9
R1525	2	3	4	3	7	**R3225**	2	4	5	2	9
R1526	1	2	1	2	4	**R3226**	1	3	2	2	6
R1527	1	2	1	2	4	**R3227**	1	3	2	2	6
R1528	1	5	5	2	10	**R3228**	1	4	5	2	11
R1531	1	2	1	2	4	**R3231**	1	2	2	1	6
R1532	1	2	3	3	9	**R3232**	1	4	3	2	9
R1533	1	2	2	2	5	**R3233**	1	4	2	1	6
R1534	1	2	2	2	5	**R3234**	2	2	3	2	7
R1535	2	6	6	3	13	**R3235**	2	6	5	2	13
R1536	1	3	1	3	5	**R3236**	1	3	2	2	7
R1537	1	2	1	2	4	**R3237**	1	2	2	1	6
R1538	1	2	2	2	5	**R3238**	1	2	3	1	6
R1611	0	3	1	1	4	**R3311**	1	2	2	1	4

Table 3.1 – Continued

Case	IS	DS	CR	Lemmas	Depth	Case	IS	DS	CR	Lemmas	Depth
R1612	0	4	1	2	5	**R3312**	1	3	1	2	4
R1613	1	3	1	2	5	**R3313**	2	2	1	2	4
R1614	1	3	1	2	5	**R3314**	2	2	1	2	4
R1615	1	4	1	2	5	**R3315**	2	4	1	2	5
R1616	0	3	1	2	4	**R3316**	1	2	1	1	4
R1617	0	3	1	2	4	**R3317**	1	2	1	1	4
R1618	1	3	1	2	5	**R3318**	2	2	1	2	4
R1621	2	2	1	2	5	**R3311**	1	3	1	2	5
R1622	1	2	3	1	6	**R3322**	1	3	1	2	5
R1623	4	5	3	2	10	**R3323**	3	5	2	2	7
R1624	8	3	3	3	11	**R3324**	2	4	2	2	7
R1625	2	4	7	2	11	**R3325**	2	4	4	2	8
R1626	1	3	1	2	5	**R3326**	1	3	2	2	5
R1627	1	3	1	2	3	**R3327**	1	4	2	2	5
R1628	2	4	4	2	8	**R3328**	2	6	4	4	10
R1631	1	2	1	1	5	**R3331**	1	3	1	2	5
R1632	3	4	3	2	10	**R3332**	1	7	2	2	8
R1633	1	2	2	1	6	**R3333**	2	2	2	2	7
R1634	1	2	2	1	6	**R3334**	2	2	2	2	6
R1635	2	6	5	2	12	**R3335**	2	4	3	2	9
R1636	1	2	1	1	5	**R3336**	1	3	1	2	5
R1637	1	2	1	1	5	**R3337**	1	3	1	2	5
R1638	1	2	2	1	6	**R3338**	2	2	2	2	6
R2111	0	3	1	1	4	**R3411**	1	2	2	1	5
R2112	0	4	1	2	4	**R3412**	1	3	2	2	5
R2113	1	3	1	2	4	**R3413**	2	2	2	2	6
R2114	1	3	1	2	4	**R3414**	3	2	2	2	6
R2115	1	5	1	2	5	**R3415**	2	4	2	2	5
R2116	0	4	1	2	4	**R3416**	1	3	2	2	5
R2117	0	4	2	2	4	**R3417**	1	2	2	1	5
R2118	1	3	1	2	4	**R3418**	2	2	2	2	6
R2121	0	3	1	1	4	**R3421**	1	3	2	2	6
R2122	0	4	2	2	4	**R3422**	1	3	2	2	6
R2123	2	5	2	2	8	**R3423**	4	5	3	2	9
R2124	1	5	2	2	6	**R3424**	3	4	3	2	9
R2125	1	6	4	2	9	**R3425**	2	4	4	2	9
R2126	0	4	1	2	4	**R3426**	1	3	2	2	6
R2127	0	4	1	2	4	**R3427**	1	3	2	2	6
R2128	1	5	2	2	5	**R3428**	1	4	5	2	10
R2131	0	3	1	1	4	**R3431**	1	3	2	2	6
R2132	0	6	2	2	6	**R3432**	1	7	3	2	9
R2133	1	3	2	2	5	**R3433**	1	2	4	1	7
R2134	1	3	2	2	5	**R3434**	2	2	5	2	9
R2135	1	7	3	2	8	**R3435**	2	5	4	2	11
R2136	0	5	1	2	5	**R3436**	1	3	2	2	7
R2137	0	4	1	2	4	**R3437**	1	2	2	1	7
R2138	1	3	2	2	5	**R3438**	1	2	4	1	7
R2211	0	7	2	1	7	**R3511**	1	2	2	1	5
R2212	0	5	2	2	7	**R3512**	1	3	2	2	5
R2213	1	4	2	2	7	**R3513**	2	2	2	2	6

Table 3.1 – Continued

Case	IS	DS	CR	Lemmas	Depth	Case	IS	DS	CR	Lemmas	Depth
R2214	1	6	2	2	7	**R3514**	2	4	2	2	7
R2215	1	9	2	2	7	**R3515**	2	4	2	2	7
R2216	0	7	2	1	6	**R3516**	1	3	2	2	5
R2217	0	7	2	1	6	**R3517**	1	2	2	1	5
R2218	1	7	2	2	8	**R3518**	2	2	2	2	6
R2221	0	9	2	2	7	**R3521**	2	2	2	2	6
R2222	0	9	3	2	7	**R3522**	1	3	4	2	8
R2223	3	4	2	1	7	**R3523**	4	5	4	2	10
R2224	4	6	3	2	10	**R3524**	3	2	2	1	6
R2225	1	9	4	2	11	**R3525**	2	5	9	2	12
R2226	0	8	2	2	8	**R3526**	1	3	2	2	8
R2227	0	9	2	2	8	**R3527**	1	3	2	2	6
R2228	0	6	3	2	9	**R3528**	1	4	9	2	12
R2231	0	4	2	2	8	**R3531**	1	3	2	2	6
R2232	1	4	2	2	8	**R3532**	1	4	4	2	10
R2233	1	4	2	2	9	**R3533**	1	2	4	1	7
R2234	2	4	4	3	9	**R3534**	1	2	4	1	7
R2235	2	11	6	3	17	**R3535**	2	6	5	2	13
R2236	1	5	2	3	8	**R3536**	1	3	2	2	7
R2237	1	4	3	2	9	**R3537**	1	2	2	1	6
R2238	1	5	4	2	10	**R3538**	1	2	4	1	7
R2311	1	4	1	1	5	**R3611**	1	2	2	1	5
R2312	1	5	1	2	5	**R3612**	1	3	2	2	5
R2313	2	4	1	2	5	**R3613**	2	2	2	2	6
R2314	2	4	1	2	5	**R3614**	2	4	2	2	7
R2315	2	6	1	2	5	**R3615**	2	4	2	2	7
R2316	0	6	1	2	5	**R3616**	1	2	2	1	5
R2317	1	4	1	1	5	**R3617**	1	2	2	1	5
R2318	2	4	1	2	5	**R3618**	2	2	2	2	6
R2321	1	5	1	2	5	**R3621**	1	3	2	2	6
R2322	1	5	1	2	5	**R3622**	1	3	4	2	8
R2323	3	7	2	2	7	**R3623**	4	5	5	2	10
R2324	2	6	2	2	8	**R3624**	3	2	2	1	6
R2325	2	6	4	2	10	**R3625**	2	4	9	2	12
R2326	1	5	1	2	6	**R3626**	1	3	2	2	6
R2327	1	4	1	1	5	**R3627**	1	3	2	2	6
R2328	3	7	4	2	10	**R3628**	1	4	7	2	12
R2331	1	3	1	2	5	**R3631**	1	3	2	2	7
R2332	1	6	2	2	8	**R3632**	1	7	4	2	10
R2333	2	4	2	2	5	**R3633**	1	2	3	1	6
R2334	2	4	2	2	5	**R3634**	1	2	4	1	7
R2335	2	7	3	2	10	**R3635**	2	9	6	2	14
R2336	1	5	1	2	6	**R3636**	1	3	2	2	7
R2337	1	4	1	1	5	**R3637**	1	2	2	1	6
R2338	2	4	2	2	7	**R3638**	1	2	2	1	5

3.2 Lazy generation of induction schemas for algorithm synthesis

By *algorithm synthesis* we understand finding an algorithm which satisfies a given specification. We are working in the context of proof-based synthesis of functional algorithms, starting from their formal specifications. A formal specification is expressed as two predicates: the input condition $I[X]$ and the output condition $O[X,T]$.[13] The desired function F must satisfy the correctness condition $\underset{X}{\forall}(I[X] \Rightarrow O[X,F[X]])$, which corresponds to the *synthesis conjecture:* $\underset{XT}{\forall\exists}(I[X] \Rightarrow O[X,T])$. An algorithm which implements F can be extracted from the (constructive) proof of this conjecture. The algorithm is expressed as a list of *clauses* (conditional rewrite rules) of the form: $C[\overline{Z}] \Longrightarrow F[P[\overline{Z}]] = \mathcal{T}[\overline{Z}]$, where $\overline{Z}$ is a vector of variables, $P[\overline{Z}]$ is a pattern over these variables (with the property that it matches unambiguously certain elements of the domain), and $\mathcal{T}[\overline{Z}]$ is a term over the matching variables. In [Dramnesc *et al.*, 2019, Dramnesc *et al.*, 2016a, Dramnesc *et al.*, 2016b, Dramnesc *et al.*, 2015a, Dramnesc *et al.*, 2015b, Dramnesc *et al.*, 2015c], we focused on developing effective and efficient techniques for mechanizing the synthesis–proofs of sorting algorithms over the domain of binary trees, the synthesis–proofs of the auxiliary functions occurring in these algorithms, and the proofs of the additional properties which are necessary in the synthesis–proofs.

Our approach is experimental: we try various scenarios and techniques and refine them in order to obtain efficient proofs and various algorithms. The way the constructive proof is built is essential since the definition of the algorithm depends on it. For example, case splits may generate conditional branches and induction steps may produce recursive definitions. Hence, the use of different case reasoning techniques and induction principles may output different algorithms. The soundness of the proof rules and of the extraction procedure guarantee the correctness of the algorithm.

The focus of our work is on proof automation. In our experiments all the proofs are produced completely automatically, while the theory exploration (identification of all necessary auxiliary statements), the selection of the assumptions and of the induction principles used by the prover in each proof, and the construction of the conjectures from the failing proofs are performed manually.

The implementations of the case studies presented in this chapther have been carried out in the frame of the *Theorema* system [Buchberger *et al.*, 2016] which is itself implemented in Mathematica [Wolfram, 2003]. *Theorema* offers significant support for automating the algorithm synthesis; in particular, the new proof strategies and inference rules have been quickly prototyped, tested and integrated in the system thanks to its extension features. Also, the proofs are easier to understand since they are presented in a human-oriented style. Moreover the synthesized algorithms can be directly executed in the system. The implementation files and the full proofs are presented in [Dramnesc *et al.*, 2015b].

Our main results include the novel synthesized algorithms: five sorting algorithms plus the auxiliary functions necessary for them: one algorithm for *Insert* (insert an element into a sorted tree), and three algorithms for *Merge* (merge two sorted trees into a sorted one).

More importantly, our experiments reveal a valuable arsenal of proof techniques (inference rules and proof strategies) which are both of general interest in natural–style proving, as well of particular interest in proof–based algorithm synthesis and in proving over the domain of binary trees.

The induction reasoning performed during the proof construction is lazy, meaning that the induction hypotheses are used by need, hence avoiding unnecessary backtracking steps. The lazy induction reasoning can be afterwards captured by explicit induction schemas such that the whole reasoning is reconstructed to an explicit induction proof which is successfully conducted at the end with the *Theorema* system. It may happen that different proof scenarios generate different induction schemas. Some of the auxiliary functions, e.g. *Merge*, use nested recursion. In this case is difficult to guess an induction principle which can be applied. Our novel combinatorial technique combined with lazy induction is able to find the appropriate induction principle, by guessing a suitable witness term.

By theory exploration in *Theorema* we produce 3 axioms, 11 definitions, and more than 200 properties. These theories are useful for the further study of properties and algorithms on binary trees. Although produced "manually", these theories can be of interest for the study of possible automation of the theory exploration process.

[13] The square brackets are used for function and predicate applications instead of round brackets.

Related Work. Algorithm synthesis and program generation is currently a challenging problem for programming and verification communities.[14] Note however that our work does not address *program generation*, which consists in transforming the description of an algorithm given in a certain formalism, into a program expressed in a certain language.

Automation of *Theory Exploration* is an active area of research – see e. g. [Colton, 2012], however in our research we did not focus on the mechanization of this process. Rather, we apply the basic principles described in [Buchberger, 2000] in order to perform top–down in parallel with bottom–up exploration, by adding the notions and properties which are necessary for our proofs.

Induction Reasoning is used successfully for algorithm synthesis – see e. g. [Bundy *et al.*, 2006, Johansson *et al.*, 2011, McCasland and Bundy, 2006]. Powerful techniques (for instance *rippling*) have been developed for overcoming a basic drawback of explicit induction: it may be that the desired algorithm cannot be constructed using the apriori given induction. Our research complements this efforts by introducing a combinatorial technique for the generation of the witness terms, as well as a lazy induction method based on Noetherian ordering, which is able to discover new explicit induction schemes.

We perform *Algorithm Synthesis* by following the classical proof–based synthesis approach as presented in e. g. [Bundy *et al.*, 2006]. The work of [Gulwani, 2010] is a comprehensive overview of the most common approaches used to tackle the synthesis problem. In [Basin *et al.*, 2004] we find a comparison between three synthesis methods: constructive/deductive synthesis, schema-based synthesis and inductive synthesis. We do not focus on the theoretical aspects of algorithm synthesis, but we follow an experimental approach in which we can find efficient proof methods. Concerning the sorting of binary trees, in classical approaches – see e. g. [Knuth, 1998] the problem of sorting them directly is not investigated, and we did not find other descriptions of such algorithms.

In the following, we present the context of binary trees and the notations, introduce the synthesis problem and describe the main synthesis method: the construction of the synthesis conjecture, its proof by induction, and the extraction of the algorithm from the proof.

General Conventions. Similar to the *Theorema* style, we use square brackets for function and for predicate application (e.g., f[x] instead of f(x) and P[a] instead of P(a)). Moreover, the quantified variables are written under the quantifier, that is $\underset{x}{\forall}$ ("for all x") and $\underset{T}{\exists}$ ("exists T"). Sometimes, the place under the quantifier also contains a property of the quantified object.

3.2.1 Sorting of binary trees

We consider *binary trees* over *elements* from a domain with a total ordering. The two types however are not explicitly present in the formulas, but only implicit by notation and by using predicate and function symbols which are not overloaded. Lower-case letters (e.g., a, b, n) represent tree elements, and upper-case letters (e.g., X, T, Y, Z) represent trees.

From the names of variables present in the original formulas, the provers generates meta–variables (denoted usually by starred symbols — e.g., T^*, T_1^*, Z^*) and Skolem constants (e.g., X_0, X_1, a_0).

The ordering between tree elements is denoted by the usual $\leq$, and the ordering between a tree and an element is denoted by: $\preceq$ (e.g., $T \preceq z$ states that all the elements from the tree T are smaller or equal than the element z, $z \preceq T$ states that z is smaller or equal than all the elements from the tree T).

We use two constructors for binary trees, namely: ε for the empty tree, and the triplet $\langle L, a, R\rangle$ for non-empty trees, where L and R are trees and a is the root element.

Predicates: $\approx$ and *IsSorted* have the following interpretations, respectively: $X \approx Y$ states that X and Y have the same elements with the same number of occurrences (but may have different structures), i.e., X is a *permutation* of Y; *IsSorted*$[X]$ states that X is a sorted tree. A tree is a *sorted* (or *search*, or *ordered*) tree if it is either ε or of the form $\langle L, a, R\rangle$ such that i) $L \preceq a \preceq R$, and ii) L and R are sorted trees.

[14] http://research.microsoft.com/en-us/um/people/sumitg/pubs/synthesis.html

3.2.2 Synthesis problem and method

As stated in the introduction, the *specification* of the target function F consists of two predicates: the input condition $I[X]$ and the output condition $O[X,T]$, and the correctness property for F is $\underset{X}{\forall}(I[X] \Rightarrow O[X,F[X]])$. The synthesis problem is expressed by the conjecture: $\underset{XT}{\forall\exists}(I[X] \Rightarrow O[X,T])$. The proof-based synthesis consists in proving this conjecture in a constructive way and then extracting the algorithm for the computation of F from this proof.

In the case of sorting, the input condition specifies the type of the input, therefore it is missing since the type is implicit. The output condition $O[X,T]$ is $X \approx T \wedge \mathit{IsSorted}[T]$ thus the synthesis conjecture becomes:

Conjecture 1 $\underset{X}{\forall}\underset{T}{\exists}(X \approx T \wedge \mathit{IsSorted}[T])$

This conjecture can be proved in several ways. Each constructive proof is different depending on the applied induction principle and the content of the knowledge base (the set of assumptions provided to the prover). Hence, different algorithms are extracted from different proofs.

3.2.3 Induction principles and algorithm extraction

The illustration of the induction principles and algorithm extraction in this subsection is similar to the one from [Dramnesc and Jebelean, 2011] for lists, but the induction principles are adapted for trees and the extracted algorithms are more complex.

The following induction principles are direct *term-based* instances of the Noetherian induction principle and can be represented using *induction schemas*. Consider the domain of binary trees with a well-founded ordering $<_t$ and denote by $\ll_t$ the multiset extension of $<_t$ as a well-founded ordering over vectors of binary trees. An induction schema to be applied to a predicate $\underset{\overline{x}}{\forall}P[\overline{x}]$ defined over a vector of tree variables $\overline{x}$ is a conjunction of instances of $P[\overline{x}]$ called *induction conclusions* that 'cover' $\underset{\overline{x}}{\forall}P[\overline{x}]$, i.e., for any value $\overline{v}$ from the domain of $\overline{x}$, there is an instance of an induction conclusion $P[\overline{t}]$ that equals $P[\overline{v}]$, where $\overline{t}$ is a vector of trees. An induction schema may attach to an induction conclusion $P[\overline{t}]$, as *induction hypotheses*, any instance $P[\overline{t'}]$ of $\underset{\overline{x}}{\forall}P[\overline{x}]$ as long as $\overline{t'} \ll_t \overline{t}$. The induction conclusions without (resp., with) attached induction hypotheses are *base* (resp., *step*) cases of the induction schema.

In the current presentation we will use the *multiset of elements* as measure values for binary trees. Checking strict ordering $E <_t E'$ between two expressions E, E' representing trees reduces to check strict inclusion between the multisets of symbols (constants and variables except ε) occurring in the expressions. This is because the expressions representing trees contain only functions which preserve the number of elements i.e., the elements of the returned tree, on the one hand, and from the arguments, on the other hand, build equivalent multisets (*Concat, Insert, Merge*).

In our experiments, we use the following induction principles for proving P as unary predicates over binary trees.

$$\textbf{Induction-1:}\ \Big(P[\varepsilon] \bigwedge \underset{n,L,R}{\forall}\big((P[L] \wedge P[R]) \Longrightarrow P[\langle L,n,R\rangle]\big)\Big) \Longrightarrow \underset{X}{\forall}P[X]$$

This is the standard structural induction based on the definition of trees.

The 'covering' property of the two induction conclusions $P[\varepsilon]$ and $P[\langle L,n,R\rangle]$ is satisfied since any binary tree is either ε or of the form $\langle L,n,R\rangle$. $P[L]$ and $P[R]$ are induction hypotheses attached to $P[\langle L,n,R\rangle]$, and it is very easy to see that their terms are smaller than the one of the induction conclusion.

In order to synthesize the sorting algorithm as a function $F[X]$, we consider the output condition $O[X,T] : \big(X \approx T \wedge \mathit{IsSorted}[T]\big)$. **Induction-1** can be applied to prove the synthesis conjecture $\underset{XT}{\forall\exists}O[X,T]$ by taking $P[X]$ as $\underset{T}{\exists}O[X,T]$.

The proof is structured as follows:

Base case: We prove $\underset{T}{\exists}O[\varepsilon,T]$. If the proof succeeds to find a ground witness $\Im_1$ such that $O[\varepsilon,\Im_1]$, then we know that $F[\varepsilon] = \Im_1$.

Step case: For arbitrary but fixed n, L_0 and R_0 (new constants), we prove $\underset{T}{\exists} O[\langle L_0, n, R_0\rangle, T]$. We assume as induction hypotheses $\underset{T}{\exists} O[L_0, T]$ and $\underset{T}{\exists} O[R_0, T]$, which are Skolemized by introducing two new constants T_1 and T_2 for each existential T. The existential quantified variable from the goal becomes the meta–variable T^* (for which we need to find a substitution term). If the proof succeeds to find a witness $T^* = \Im_2[n, L_0, R_0, T_1, T_2]$ (term depending on n, L_0, R_0, T_1 and T_2), then we know that $F[\langle L, n, R\rangle] = \Im_2[n, L, R, F[L], F[R]]$.[15]

The extracted algorithm from the proof is expressed as:

$$\underset{n,L,R}{\forall} \left(\begin{array}{c} F[\varepsilon] = \Im_1 \\ F[\langle L, n, R\rangle] = \Im_2[n, L, R, F[L], F[R]] \end{array} \right)$$

This function definition expressed as two equalities can be easily transformed into a functional program by using appropriate decomposition functions which extract the root, the left branch, and the right branch from the tree.

The theoretical basis and the correctness of this proof-based synthesis scheme is well known – see for instance [Bundy *et al.*, 2006].

For the following induction principles, the proof and the algorithm extraction are similar to **Induction-1**, therefore we give only the structure of the extracted algorithm for each induction principle.

Induction-2:

$$\Big(P[\varepsilon] \bigwedge \underset{n,L}{\forall} \big(P[L] \Longrightarrow P[\langle L, n, \varepsilon\rangle]\big) \bigwedge \underset{n,L,R}{\forall} \big((P[\langle L, n, \varepsilon\rangle] \wedge P[R]) \Longrightarrow P[\langle L, n, R\rangle]\big)\Big) \Longrightarrow \underset{X}{\forall} P[X]$$

This induction principle was suggested by the experiments, according to the induction schema discovery explained in Subsection 3.2.4, and illustrated at the end of that subsection.

The extracted algorithm is:

$$\underset{n,L,R}{\forall} \left(\begin{array}{c} F[\varepsilon] = \Im_1 \\ F[\langle L, n, \varepsilon] = \Im_3[n, L, F[L]] \\ F[\langle L, n, R\rangle] = \Im_5[n, L, R, F[\langle L, n, \varepsilon\rangle], F[R]] \end{array} \right)$$

Induction-3:

$$\begin{array}{c} \Big(P[\varepsilon] \bigwedge \underset{n}{\forall}(P[\langle \varepsilon, n, \varepsilon\rangle]) \bigwedge \underset{n,L}{\forall}(P[L] \Longrightarrow P[\langle L, n, \varepsilon\rangle]) \bigwedge \underset{n,R}{\forall}(P[R] \Longrightarrow P[\langle \varepsilon, n, R\rangle]) \bigwedge \\ \underset{n,L,R}{\forall}((P[L] \wedge P[R]) \Longrightarrow P[\langle L, n, R\rangle])\Big) \\ \Longrightarrow \underset{X}{\forall} P[X] \end{array}$$

This is an expression of Induction-1 which makes explicit the cases of empty subtrees, also discovered experimentally using the induction schema discovery. L and R are assumed to be non-empty trees. In order to encode this conveniently during the proof, they are replaced by $\langle A, a, B\rangle$ and $\langle C, b, D\rangle$, respectively.

The extracted algorithm is:

$$\underset{n,a,b,A,B,C,D}{\forall} \left(\begin{array}{c} F[\varepsilon] = \Im_1 \\ F[\langle \varepsilon, n, \varepsilon\rangle] = \Im_2[n] \\ F[\langle\langle A, a, B\rangle, n, \varepsilon\rangle] = \Im_3[n, A, a, B, F[\langle A, a, B\rangle]] \\ F[\langle \varepsilon, n, \langle C, b, D\rangle\rangle] = \Im_4[n, C, b, D, F[\langle C, b, D\rangle]] \\ F[\langle\langle A, a, B\rangle, n, \langle C, b, D\rangle\rangle] = \Im_5[n, a, b, A, B, C, D, \\ F[\langle A, a, B\rangle], F[\langle C, b, D\rangle]] \end{array} \right)$$

3.2.4 Refining induction by lazy reasoning

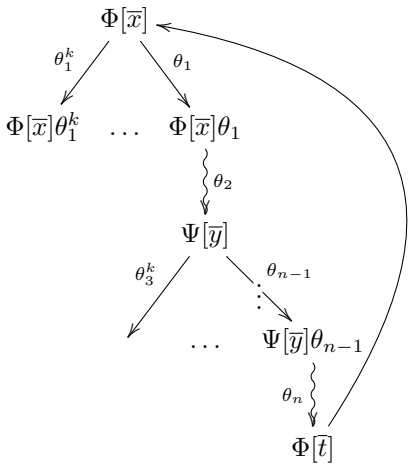
As shown in the general introduction, sometimes the concrete induction principle which is used for proving does not succeed. In this case, one needs to think about a more powerful principle and reiterate the proof

[15] T_1 and T_2 are replaced by $F[L]$ and $F[R]$, respectively.

attempt. We present here a technique able to find automatically and in a lazy way, during the proof, new term-based Noetherian induction principles which are necessary.

The technique helps to prove a formula $\underset{\overline{x}}{\forall}\Phi[\overline{x}]$ by lazy induction, where $\overline{x}$ is a vector of variables. Firstly, we start to instantiate variables from $\overline{x}$, then transform the resulted instances by using deductive rules. The instantiation and deduction steps can be intertwined up to the moment when instances of $\Phi[\overline{x}]$ are encountered. Finally, an instance $\Phi[\overline{t}]$ can be used as induction hypothesis for the induction case $\Phi[\overline{x}]\theta$ if $\overline{t} \ll_t \overline{x}\theta$, where the vectors $\overline{t}$ and $\overline{x}\theta$ are represented as multisets of terms. This is a particular case of the formula-based Noetherian induction method presented in Chapter 2 for which the cyclic proofs have only 1-cycles and the IHs are instances of the root formula.

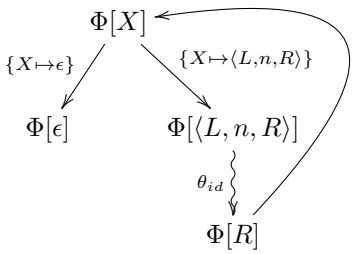
The *cumulative* substitution θ is built from the proof. To illustrate its computation, we represent the proof derivation as a tree for which the root node is labeled by $\Phi[\overline{x}]$. Two kinds of non-root nodes are distinguished: instantiation nodes and deductive nodes. The instantiation nodes are direct successors of a node N labeled by a formula with free variables for which some of them are instantiated with terms whose variables are fresh. The set of instance formulas labeling all the instantiation nodes should *cover* the formula labeling N and can be built from the sort of the instantiated variables. For example, if N is labeled by the formula $\Phi[X]$, a covering set of instance formulas is $\{\Phi\{X \mapsto \epsilon\}, \Phi\{X \mapsto \langle L, n, R\rangle\}\}$, where L, n, R are fresh variables. In the graphical representation of a proof tree, the relation between a node and its direct instantiation nodes are represented by downward solid arrows annotated by the corresponding instantiation substitution. The deductive nodes are direct successors of nodes to which a deductive operation has been applied. These relations are graphically represented as curly arrows annotated by identity substitutions. The cumulative substitution is the composition of the substitutions annotating the nodes from the path leading from the root node, in our case the node labeled by $\Phi[\overline{x}]$, to the node labeled by the induction hypothesis, in our case $\Phi[\overline{t}]$. This scenario can be illustrated as below:



In our scenario, $\Phi[\overline{t}]$ is an instance of $\Phi[\overline{x}]$. In addition, it can be used as an induction hypothesis if $\overline{t}$ is smaller than $\overline{x}\theta_1\theta_2\cdots\theta_{n-1}\theta_n$.

Example 10 *By lazy induction, one can benefit of more effective induction reasoning, involving only useful induction hypotheses.*

Let us assume the following scenario for processing a formula $\Phi[X]$, where X is a binary tree:

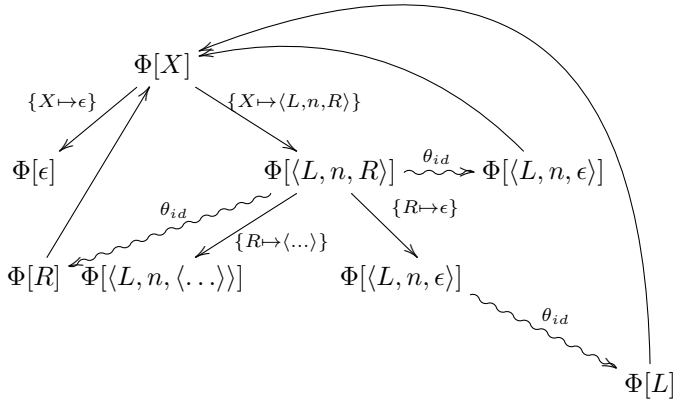


Here, θ_{id} is the identity substitution $\{L \mapsto L; n \mapsto n; R \mapsto R\}$. $\Phi[R]$ can be used as induction hypothesis in the proof of the case $\Phi[\langle L, n, R\rangle]$ because R has a number of elements smaller than $\langle L, n, R\rangle$.

The corresponding explicit induction principle is:

$$\Big(P[\varepsilon] \bigwedge \underset{n,L,R}{\forall} \big(P[R]) \Longrightarrow P[\langle L, n, R\rangle]\big)\Big) \Longrightarrow \underset{X}{\forall} P[X]$$

Example 11 *More specific induction schemas can also be generated by lazy induction, as shown in the following scenario:*



The corresponding explicit induction principle is:

$$\Big(P[\varepsilon] \bigwedge \underset{n,L}{\forall} \big(P[L] \Longrightarrow P[\langle L, n, \varepsilon\rangle]\big) \bigwedge \underset{n,L,R}{\forall} \big((P[\langle L, n, \varepsilon\rangle] \wedge P[R]) \Longrightarrow P[\langle L, n, R\rangle]\big)\Big) \Longrightarrow \underset{X}{\forall} P[X]$$

3.3 Conclusions

We have defined a set of conjectures that can be proved using cyclic induction by following a similar scenario and based on the variable instantiation schema of the Peano induction. We have shown how to implement the cyclic induction reasoning in Coq, by means of external libraries and functional schemas issued from the definition of new function symbols. The associated explicit induction principles, automatically generated by Coq, are more complex than the Peano induction principle. On the other hand, all but three conjectures have also been proved by Peano induction but their proofs are *ad hoc* and do not follow the cyclic proof script. Indeed, they are very different in terms of scenario, length and difficulty.

The procedure for generating explicit induction principles from cyclic induction reasoning was successfully used for the synthesis of algorithms for binary trees. It led to discovering new recursion structures that are not given from the beginning by the induction principle, new induction schemas and new algorithms with nested recursion.

Part II

Mechanising and Certifying Noetherian Induction Reasoning

4

Tool: the SPIKE Prover

Sommaire

4.1 The inference system and proof strategies 46

4.1.1 The 'reasoning by implicit induction' setting 46

4.1.2 The inference system 46

4.1.3 The layout of a specification file 49

4.2 A complete example of SPIKE specification 50

4.3 Conclusions 52

SPIKE, an induction-based theorem prover built to reason on conditional theories with equality, is one of the few formal tools able to perform automatically mutual and lazy induction. Designed at the beginning of 1990s, it has been successfully used in many non-trivial applications and served as a prototype for different proof experiments and extensions. The first paper introducing SPIKE is [Bouhoula *et al.*, 1992], published shortly after the tool was created. The goal of this chapter is to highlight and bring together in one spot the major changes supported by SPIKE since then.

Context. Historically, SPIKE was built during 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 designed. Some of them have been implemented into theorem provers, for example, RRL [Kapur and Zhang, 1988] integrated the test-set induction method [Kapur *et al.*, 1986], and Focus [Bronsard and Reddy, 1991] a generalization of the term-rewriting induction [Bronsard *et al.*, 1994] for conditional theories. Inspired by the rewriting techniques previously tested with the ORME system [Lescanne, 1990], SPIKE [Bouhoula *et al.*, 1992] implemented a different induction method [Kounalis and Rusinowitch, 1990b, Bouhoula *et al.*, 1995] that combines features from explicit induction and inductive completion techniques.

As time went by, SPIKE was continuously considered among the 'active' automatic induction-based provers; it mainly served as a prototype for testing several extensions of conditional theories and induction-based reasoning techniques that led to many successful proof experiments on non-trivial applications. In this chapter, we highlight the major changes supported by SPIKE since the publication of [Bouhoula *et al.*, 1992], which gave rise to its current version. The source code, examples of specification files, and papers related to different applications and extensions are available online [The SPIKE development team, 2020].

The content of this chapter is based on the paper [Stratulat, 2020]. In the rest of the chapter, we set the theoretical backgrounds of the reasoning by induction on equational clauses, then we introduce the inference system of SPIKE and the layout of a standard specification file. In the end, we show how to prove conjectures and interact with the tool.

4.1 The inference system and proof strategies

In this section, we first present the general setting for proving by implicit induction, then introduce the inference system of SPIKE followed by the layout of specification files. Finally, we present the different ways the user can interact with the tool.

4.1.1 The 'reasoning by implicit induction' setting

SPIKE implements an instance of the formula-based Noetherian induction principle applied on a Noetherian poset of equational clauses (or just clauses). Several clauses can be simultaneously tested to verify whether they are consequences of a given set of axioms Ax written as conditional equalities. SPIKE can reason on sorted and constructor-based specifications that should satisfy some properties, like the ground convergence and (strongly sufficient) completeness [Bouhoula, 1997]. These constraints guarantee the existence of the *initial model* for Ax. Formally, assuming that $\mathcal{M}$ is the initial model of Ax, we denote by $\Phi \models_{\mathcal{M}} \phi$ the fact that the clause ϕ is an initial $\mathcal{M}$-consequence (or just *consequence*) of a set of conditional equalities Φ. ϕ is initially $\mathcal{M}$-valid (or just valid) and denoted by $\models_{\mathcal{M}} \phi$ iff it is a consequence of Ax.

The induction proof method used by SPIKE, called *implicit induction* and detailed in Chapter 1, is based on reductive techniques as rewriting. By implementing the formula-based Noetherian induction principle, it can naturally perform lazy and mutual induction steps. The mutual induction feature helped SPIKE to prove the Gilbreath Card trick problem [Huet, 1991], firstly with 5 lemmas [Bouhoula and Rusinowitch, 1993] then with only 2 lemmas [Bouhoula and Rusinowitch, 1995], while other provers using different proof techniques succeeded with significantly more lemmas (see [Bouhoula and Rusinowitch, 1995] for a comparison).

The ground convergence and completeness properties of a specification can be checked more easily, by using syntactic criteria, if the specification is *many sorted* and the set of function symbols is split into *constructor* and *defined function* symbols. SPIKE was initially designed to deal with *free* constructors such that there is no equality relation between any two different constructor symbols. Several extensions have been introduced in SPIKE since [Bouhoula *et al.*, 1992] in order to deal with: i) non-free constructors [Bouhoula and Jouannaud, 2001], ii) parameterized specifications [Bouhoula, 1994b, Bouhoula, 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.

4.1.2 The inference system

In [Bouhoula *et al.*, 1992], the inference rules and the proof strategy implementing the implicit induction method were built-in. Each rule is a transition between pairs (E, H), where E are *conjectures* and H are *premises* consisting of previously processed conjectures that do not have minimal *counterexamples*, i.e., minimal ground clauses that are not valid. By applying a rule, a conjecture from the current proof state is replaced by a potentially empty set of new conjectures, and may be added as a premise in order to participate to further inference steps. Proof *derivations* are built by successively applying inference rules starting from an initial state. They may finish with i) *success*, if they end with an empty set of conjectures, ii) *error*, if a counterexample is detected, and iii) *failure*, if none of the previous cases is encountered and no rule can be applied. A *proof* is a successful derivation that starts with an empty set of premises.

Later on, different proof needs led to hardcode into the system several variants of a same rule. More flexibility has been achieved with the addition of a strategy language [Alouini and Bouhoula, 1997] allowing to define new proof strategies by the user. It has been combined later with a methodology for building modular inference rules using *contextual cover sets* (CCSs), detailed later in Section 8.1. The core of the methodology is the abstract inference system A, defined and proved sound in [Stratulat, 2001]. A is similar to the inference system I from Chapter 1 and is made of two rules: AddPremise and Simplify,

defined as:

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 $\psi \leq \phi\tau$.

Each inference rule replaces a conjecture ϕ with a potentially empty set of new conjectures Φ. Φ is built in two steps as a CCS of ϕ, thanks to the compositional properties of CCSs. Firstly, an intermediate CCS of ϕ is built, then for each intermediate clause another CCS is built and stored as new conjectures. The set of IHs allowed by a rule to be used when building a CCS is referred to as *context*. ADDPREMISE adds ϕ as a premise and SIMPLIFY allows bigger contexts. It has been shown in [Stratulat, 2001] that i) the abstract inference system is *sound*, i.e., one can conclude from a proof that its initial conjectures are valid, and ii) the inference rules define the biggest contexts compared to similar abstract rules proposed in the literature. For practical reasons, the abstract system was extended with a third rule, DELETE, that is a particular case of SIMPLIFY when Φ is empty.

Any SPIKE inference rule instantiates one of the abstract rules by implementing its elementary CCSs, i.e. the CCSs that are not built using composition operations, by the means of *reasoning modules*. A reasoning module can produce a CCS with a particular reasoning technique using in terms of IHs clauses from the context defined by the instantiated abstract rule. The main reasoning techniques are based on rewriting, case analysis and variable instantiations.

We give as example the definition of a rewriting-based inference operation implemented as an instance of SIMPLIFY:

```
rewriting_rule = simplify(id,[rewriting(rewrite,L|R|C,*)]);
```

In the first step, the identity reasoning module `id` builds $\{\phi\}$ as the intermediate CCS for ϕ. The application of the rule succeeds if, in the second step, the `rewriting` module succeeds to rewrite once, due to the `rewrite` argument and at any position (`*`), the only clause ϕ of $\{\phi\}$ with conditional rewrite rules from the lemmas (`L`), axioms (`R`) and current context (`C`). The resulting clause is the unique clause of Φ.

Some of the reasoning techniques have been changed since [Bouhoula *et al.*, 1992]. For example, the technique for instantiating variables with elements of a test-set, based on the depth of the lhs of the axioms [Bouhoula and Rusinowitch, 1995], was replaced by a narrowing technique involving only the axioms defining the head symbol of some (sub)term including the variables to be instantiated [Barthe and Stratulat, 2003].

New reasoning techniques have been added to deal with non-trivial applications. The implementation of a combination between a decision procedure for linear arithmetic and a congruence closure algorithm [Stratulat, 2000b, Armando *et al.*, 2002] allowed 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. This implementation, as well as the 'black-box' integration schema of the Z3 [De Moura and Bjørner, 2008] SMT solver as a reasoning module, are described in [Stratulat, 2014]. SPIKE also includes several decision procedures for proving inductive theorems without induction reasoning [Aoto and Stratulat, 2014]. They help to decide the inductive validity of equations involving natural numbers and lists.

The validation of the JavaCard platform [Barthe and Stratulat, 2003], detailed in Chapter 5, was the most challenging case study ever experienced by SPIKE. The inference system has been adapted to manage variables of parameterized sorts as well as existential variables. New inference rules have been designed to better handle the information from the conditional part of (conditional) conjectures, like i) the *auto simplification rule* in order to rewrite with an equality condition other parts of the conjecture,

and ii) the *augmentation rule* which adds new conditions issued from the conclusion of a conditional equality given as lemma if the conditions of the lemma are proved from the conditions of the conjecture. 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.

As shown in [Bronsard *et al.*, 1994], the implicit induction method is based on a unique induction ordering, globally defined over the set of clauses derived during a proof. It is implemented by *reductive* inference systems such that, at the ground level, the new conjectures from any proof step are smaller than (and sometimes equal to) the replaced conjecture. The reductive techniques, as rewriting, introduce new ordering constraints to be satisfied by the specification as well as the conjectures from the proof derivation. The induction ordering is the multiset extension of the mpo ordering [Baader and Nipkow, 1998] over terms using a precedence over the function symbols provided by the user. The mpo ordering also serves to orient the axioms into rewrite rules. Since some reasoning techniques, like the instantiation of variables from the current conjecture, require the reduction of the instances by rewriting, SPIKE warns the user if the mpo ordering built from the input precedence cannot orient the axioms into rewrite rules. If no precedence is provided by the user, SPIKE analyses the axioms and tries to infer a successful precedence.

The inference system of SPIKE has been extended to implement for the first time reductive-free cyclic proofs [Stratulat, 2012], also described in Chapter 2, by keeping the best features of explicit and implicit induction reasoning. To recall, a *cycle* consists of a circular linked list of proof derivation chunks, called *history chunks*. Each link symbolises the application of an instance of the head conjecture from a history chunk as IH in the proof of the conjecture ending the previous history chunk in the list. Following the *DRaCuLa* proof strategy, the cycle can discharge its linking IHs by checking ordering constraints involving only instances of the conjectures starting the history chunks. Therefore, the cyclic induction reasoning allows to use non-reductive proof techniques along history chunks and axioms not orientable into rewrite rules as long as the ordering constraints are satisfied.

A useful property for an inference system is the *refutational soundness*, which guarantees that, whenever a counterexample is detected at the current step, the initial conjectures are refuted. Very few inference rules implemented by SPIKE may add new counterexamples during the proof derivation, e.g., by the generalisation of existential into universal variables [Barthe and Stratulat, 2003]. By attaching history information to each conjecture, the detected counterexamples can lead to particular ground instances of the initial conjectures that can be further checked for validity.

The *refutational completeness* is another useful property, satisfied by previous inference systems [Bouhoula *et al.*, 1995, Bouhoula, 1997]. Since the addition of a strategy language, this property is no longer guaranteed because the *fairness* of the inference system, forbidding persistent conjectures along its derivations, may be broken. However, the user still can use some built-in strategies known to be refutationally complete. A classical proof strategy mainly privileges the inference rules that are instances of DELETE, then SIMPLIFY, and finally ADDPREMISE. We give as example the definition of the `fullind` 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
decomposition_rule = simplify(id, [negative_decomposition]);
total_case_rewriting_rule = simplify(id, [total_case_rewriting
  (simplify_strat, R, *)]);

% instances of AddPremise
case_rewriting = add_premise(total_case_rewriting(simplify_strat, R, *),[id]);
split_rule = add_premise(generate,[id]);

% proof strategies
```

```
stra = repeat (try (tautology_rule,
  negative_clash_rule, subsumption_rule,
  decomposition_rule, rewriting_rule,
  print_goals, case_rewriting ));
fullind = (repeat(stra, split_rule),
  print_goals_with_history);

start_with: fullind
```

The reasoning module `total_case_rewriting`, implementing the conditional rewriting technique, was used to build instances of both SIMPLIFY and ADDPREMISE. `try` and `repeat` take a list of rules as arguments. `try` visits each rule in the list until the first succeeds. `repeat` executes them repeatedly until no new conjecture is produced or a counterexample is found. The `print_*` rules print the current state of the proof. `start_with` points to the used strategy.

4.1.3 The layout of a specification file

The structure of a standard SPIKE specification from the 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 precedencies 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 priority over the head symbols used by the instantiation technique
`conjectures:` *list of conjectures*

User interactions. The proofs, generated by the command `spike_bc` *name*.`spike`, are highly automatic, the user interactions mainly 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.

On the other hand, the generated proofs may involve many non-trivial inference steps for which the human checking process is tedious and error-prone. We have shown how to i) validate SPIKE proofs [Stratulat, 2010, Stratulat and Demange, 2011] using the certification environment provided by the Coq system [The Coq development team, 2020], and ii) define Coq tactics that call directly SPIKE and transform the generated proofs into Coq scripts [Henaien and Stratulat, 2013], according to a methodology that automatically translates into Coq script the proof steps performed by most of its inference rules.

The user can also interact with SPIKE by the means of i) extra sections, for example `use: nats;`[16] 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`, as:

`-debug`: to identify syntactic errors in the specification file,

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

`-maximal`: to print the proof in detail,

`-coqc_spec` and `-coqc`: to generate the Coq specification, and translate the generated proof into Coq script, respectively, and

`-dracula`: to generate cyclic proofs using the DRaCuLa strategy.

Coding languages. SPIKE was initially written in the 'light' version of the Caml [Cousineau and Mauny, 1998] functional language by Adel Bouhoula during his PhD thesis [Bouhoula, 1994a]. SPIKE has been completely redesigned for adopting an object-oriented paradigm along the $\sim$ 18500 lines of OCaml [Leroy *et al.*,] code.

Some of the original features are still missing, like the graphical interface and the procedures for checking the completeness and ground convergence properties.

4.2 A complete example of SPIKE specification

We provide the SPIKE specification of different definitions of 'even' and 'odd' operations that have been tested with the DRaCuLa strategy, as described in Chapter 2.

```
specification: even2odd_unoriented

sorts: nat  bool;

constructors :

0    :     -> nat;
S_   : nat -> nat;
True : bool;
False: bool;

defined functions:

even1_ : nat -> bool;
odd1_  : nat -> bool;
even_  : nat -> bool;
odd_   : nat -> bool;
plus__ : nat nat -> nat;

axioms:

plus(0, x) = x;
plus(S(x), y) = S(plus(x,y));

even(0) = True;
even(S(0))  = False;
even(S(S(x))) = even1(plus(S(S(x)),0)); % not orientable

even1(0) = True;
even1(S(0))  = False;
odd(x) = False => even1(S(S(x))) = even(x);
odd(x) = True => even1(S(S(x))) = False;

odd(0) = False;
odd(S(0)) = True;
odd(S(S(x))) = odd1(plus(S(S(x)),0)); % not orientable

odd1(0) = False;
```

```
odd1(S(0)) = True;
even(x) = True => odd1(S(S(x))) = odd(x);
even(x) = False => odd1(S(S(x))) = True;

greater:

plus : S 0;
even : True False S 0 plus;
even1: True False;
odd  : True False S plus;
odd1 : True False;

equiv: even even1 odd odd1;
properties:

system_is_sufficiently_complete ;
system_is_strongly_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
decomposition_rule =
   simplify(id, [negative_decomposition]);
rewriting_rule =
   simplify(id,[rewriting(rewrite, L|R|C, *)]);
total_case_rewriting_rule =
   simplify(id, [total_case_rewriting (
      simplify_strat, R, *)]);

% instances of AddPremise
case_rewriting =
  add_premise(total_case_rewriting(
     simplify_strat, R, *),[id]);
split_rule = add_premise(generate,[id]);

% proof strategies
stra = repeat (try (tautology_rule,
                    negative_clash_rule,
                    subsumption_rule,
                    decomposition_rule,
                    rewriting_rule,
                    print_goals,
                    case_rewriting ));
fullind = (repeat(stra, split_rule),
  print_goals_with_history);

start_with: fullind

conjectures:
```

```
plus(x, S(y)) = S(plus(x,y));

conjectures:

even(plus(x,x)) = True;
odd(plus(x,x)) = False;
```

4.3 Conclusions

We have given an overview of the current version of SPIKE and highlighted the main changes and extensions since [Bouhoula *et al.*, 1992]. Additional information about SPIKE and its applications can be found on the website [The SPIKE development team, 2020].

5

Application: Validation of the JavaCard Platform

Sommaire

5.1 A primer on CertiCartes 54
5.2 Improvements of SPIKE 55
5.2.1 At the specification language 55
5.2.2 At the proof engine 56
5.3 Applications to JavaCard 58
5.3.1 CDO 58
5.3.2 CDA and MON 58
5.3.3 Assessment 58
5.4 Conclusions and related works 61

The bytecode verifier (BCV), which performs a static analysis to reject potentially insecure programs, is a key security function of the Java(Card) platform. Over the last few years there have been numerous projects to prove formally the correctness of bytecode verification, but relatively little effort has been made to provide methodologies, techniques and tools that help such formalisations. [Barthe *et al.*, 2001b, Barthe *et al.*, 2002b] developed a methodology and a specification environment featuring a neutral mathematical language based on conditional rewriting, that considerably reduce the cost of specifying virtual machines.

In this chapter, based on [Barthe and Stratulat, 2003], we show that such a neutral mathematical language based on conditional rewriting is also beneficial for performing automatic verifications on the specifications, and illustrate in particular how implicit induction techniques can be used for the validation of the Java(Card) Platform. More precisely, we report on the use of SPIKE, presented in Chapter 4, to establish the correctness of the BCV. The results are encouraging, as many of the intermediate lemmas required to prove the BCV correct can be proved with SPIKE.

Settings. Virtual machines, such as the Java(Card) Virtual Machine, provide a means to ensure security of mobile code, because the virtual machine controls the interaction between the applet and its environment and hence reduces the risk of malicious applets performing a security attack. Furthermore, such architectures rely on several mechanisms, known as security functions. A crucial such security function of the Java(Card) architecture is the bytecode verifier which performs a static analysis on programs and rejects potentially insecure programs.

Over the last few years there have been numerous projects to specify such virtual machines and their bytecode verifiers, and to prove the correctness of bytecode verification. While several projects have been very successful in their work, such endeavours are labour-intensive and suffer from the lack of adequate tool support, see the related works from Section 5.4. Our line of work is precisely to develop methodologies, techniques and tools that reduce the cost of developing and maintaining such formalisations.

CertiCartes In early work [Barthe *et al.*, 2001b, Barthe *et al.*, 2002b], we have developed a robust methodology to validate bytecode verifiers. The methodology consists in defining three virtual machines:

- a reference, so-called defensive, virtual machine where values are tagged by their type and where typing information is verified at run-time;

- an abstract virtual machine that manipulates types and that is used for bytecode verification;
- a "standard", so-called offensive, virtual machine, where values are untyped, and which relies on successful bytecode verification to eliminate type verification at run-time.

The advantages of our methodology is three-fold:

1. the offensive and abstract virtual machines can be derived from the defensive virtual machine using abstraction techniques;
2. the correctness of bytecode verification is now crisply stated as: "the offensive and defensive virtual machines coincide on those programs that pass bytecode verification";
3. the correctness of bytecode verification follows from the correctness of the abstractions of the defensive virtual machine into an offensive and an abstract virtual machine respectively, and from a generic development that establishes the correctness of the derivation of the bytecode verifier from the abstract virtual machine—the development is presented in [Nipkow, 2001] and further refined in [Barthe and Dufay, 2003, Klein, 2003].

Jakarta In previous work [Barthe *et al.*, 2002a, Barthe *et al.*, 2001a], we argue that a neutral mathematical language is beneficial for performing automatic transformations on the specifications. Further, we introduce the Jakarta Specification Language (JSL), a simple typed specification language based on conditional rewriting, and the Jakarta Transformation Kit (JTK), an abstraction engine which constructs an offensive and an abstract virtual machine from the defensive virtual machine. The results with the JTK are encouraging, as it automates to a large extent the derivation of the offensive and abstract virtual machine; indeed, user interaction is limited to abstraction scripts that contain information on how the abstractions are to be constructed, and that are typically 10 times shorter than the latter.

Structure of the chapter. The chapter has four sections. Section 5.1 provides the necessary background on CertiCartes and introduces the problem to be addressed. Section 5.2 describes the main improvements that were implemented in SPIKE to handle the specification and validation of virtual machines. In Section 5.3, we turn to the application of SPIKE for proving the cross-validation of virtual machines. Section 5.4 discusses about related works.

5.1 A primer on CertiCartes

CertiCartes is an in-depth feasibility study in the formal verification, using the Coq proof assistant [The Coq development team, 2020], of the JavaCard Platform—recall that JavaCard is a dialect of Java tailored towards programming multi-application smartcards. In a nutshell, CertiCartes contains formal specifications of (one-step execution of) a defensive, an abstract and an offensive JavaCard Virtual Machine (JCVM) and of the BCV, and a proof that the defensive and offensive VMs coincide on those programs that pass bytecode verification.

Virtual Machines In order to formalize the semantics of the virtual machines:

- we model programs as a triple consisting of a list of classes, a list of interfaces, and a list of methods. Classes, interfaces and methods are represented likewise as appropriate tuples;
- we define for each machine a notion of state: `dstate` which builds upon typed values for the defensive machine, `astate` which takes types as values for the abstract machine, and `ostate` which builds upon untyped values for the offensive machine. Further, we define an associated notion of return states: `drstate` for the defensive machine, `arstate` for the abstract machine, and `orstate` for the offensive machine, that extends states with a tag to account for normal/abnormal termination, and in the case of the abstract machine returns lists of states to account for non-determinism;
- we model the semantics of each JavaCard bytecode `b` as a function `?exec_b: ?state` → `r?state`, where `?` ranges over `d`, `a` and `o`—note that in our formalisation, the JCVM instruction set is factorized into 45 generic instructions, as many instructions only differ by the type of their arguments, and can be factorized using a polymorphic instruction. Typically, the function `?exec_b` extracts values from the state, performs type verification on these values, and extends/updates the state with the results of executing the bytecode;
- we model one-step execution as a function `?exec: ?state` →`r?state` which inspects the state to extract the JavaCard bytecode `b` to be executed and then calls the corresponding function `?exec_b`.

In order to prove the correctness of the BCV, we must prove three crucial properties about virtual machines:

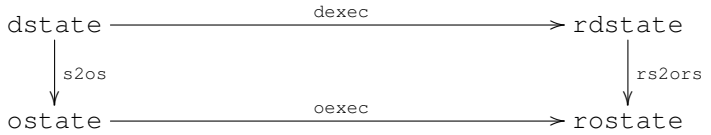


Figure 5.1: Commutative diagram of defensive and offensive execution.

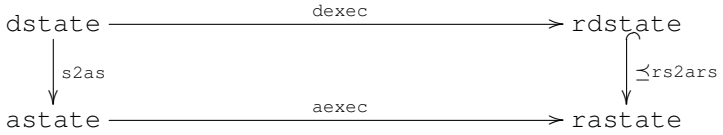


Figure 5.2: Commutative diagram of defensive and abstract execution.

- **CDO:** the offensive abstract virtual machine is a sound abstraction of the defensive virtual machine, as illustrated by the commuting (up to the absence of typing error in the defensive execution) diagram in Figure 5.1, where `s2os` is the function mapping states to offensive states (by omitting types from values), and `rs2ors` denotes its lifting to return states;
- **CDA:** the abstract virtual machine is a sound abstraction of the defensive virtual machine, as illustrated by the commuting (up to subtyping, as indicated by the $\preceq$ relation in the right arrow, and under suitable conditions, e.g. that execution does not raise an exception and keeps in the same frame) diagram in Figure 5.2, where `s2as` is the function mapping states to abstract states (by projecting values to types), and `rs2ars` denotes its lifting to return states;
- **MON:** the abstract virtual machine is monotonic w.r.t. the order induced by the inheritance relations on classes and interfaces.

For each of the properties considered, the proof proceeds by a case analysis on the bytecode to be executed, and then by an analysis of the possible outcomes of execution.

Bytecode verifier The BCV is derived by instantiating a dataflow analyser with the abstract JCVM, and its correctness is derived from CDO, CDA and MON, using a generic (i.e. independent from the specifics of the JCVM) proof that justifies the dataflow analysis and the compositional, method-by-method, algorithm underlying bytecode verification, see [Barthe and Dufay, 2003, Klein and Nipkow, 2003, Nipkow, 2001].

5.2 Improvements of SPIKE

SPIKE provides an environment to verify clausal formulas in the initial model of many-sorted constructor-based theories presented by first-order conditional rules, and hence seems to be a good candidate for proving CDA, CDO and MON. Nevertheless, the standard distribution of SPIKE could not be used, since its specification language is too restricted, and its proof engine is not sufficiently optimized. Below we report on a number of improvements that were undertaken in order to apply SPIKE to our problem.

5.2.1 At the specification language

Parameterized Specifications The JavaCard VMs specifications are based on an important number of parameterized datatypes and functions. In particular, polymorphic lists are used intensively in the memory model: for example the heap is described as a list of objects, and the stack as a list of frames; further, each frame comes up with an operand stack and a set of local variables, each of which is described as a list of values. However, such parameterized specifications are not handled by the standard version of SPIKE. In order to perform our case study, we had to extend the syntax, type checking and inference system of SPIKE to deal with parameterized

specifications—a similar work is described in detail in [Bouhoula, 1996], but had not been integrated in the standard version of SPIKE.

Introduction of Existential Variables The axioms of standard SPIKE specifications consist in conditional rewrite rules of the form

$$l_1 = r_1, \ \ldots \ , l_n = r_n \Rightarrow g \rightarrow d$$

where all variables in conditions and d are required to occur in g. Formally, SPIKE requires that $\mathsf{var}(d) \subseteq \mathsf{var}(g)$ and that for $1 \leq i \leq n$, $\mathsf{var}(l_i) \subseteq \mathsf{var}(g)$ and $\mathsf{var}(r_i) \subseteq \mathsf{var}(g)$. However, most functions defining the semantics of the JCVM fail to meet this requirement, as variables in the r_is are fresh. In order to handle the JCVM specifications, we have enhanced SPIKE with the ability to handle such variables, which we call *existential*.

Obtaining SPIKE specifications of the JCVMs We have implemented mechanisms that may be used to compile a large class of Coq specifications to JSL and SPIKE, and that have been used to produce SPIKE specifications of the JCVMs from CertiCartes.

5.2.2 At the proof engine

We only provide a concise and informal description of the extensions that we have implemented, and briefly indicate their impact on soundness.

Adaptation of the Inference System

In order to handle the extensions of the specification language, we have modified the inference system, and in particular the Generate rule, an instance of the abstract rule AddPremise from Chapter 4. Similar to the ConjSup rule from Chapter 1, it instantiates variables and rewrites with conditional rewrite rules the clause instances using narrowing techniques. First, the parameterized variables cannot be instantiated by Generate rules during the proofs. Second, the Generate rule is modified so that SPIKE i) forbids the instantiation of existential variables, unless all induction variables are tagged as existential; ii) does not put the current conjecture in the set of premises if its cover set instances are simplified with conditional rewrite rules introducing existential variables. W.r.t. i), observe that if no special provision were made, no inference rule could be applied if all induction variables are tagged as existential. In some circumstances however, we may want the proof to proceed, and so we force such a behavior by generalizing existential variables to universal ones in order to perform Generate. One drawback of this solution is the loss of refutational soundness, as the new rule potentially introduces new artificial counterexamples in the derivation by a generalization operation. Nevertheless, the new rule preserves the soundness of the system. W.r.t. ii), existential variables break the order condition requiring that the left hand side be greater than the conditions and the right hand side. This implies that if the current conjecture would contain a counterexample, the new set of conjectures cannot guarantee a smaller one. However, this condition is crucial for allowing the current conjecture to participate to further inferences [Stratulat, 2001]. In such cases, the transformation of a cover set instance $C\sigma$ can be considered as an instance of Simplify rule from Chapter 4, which in addition (w.r.t. the Generate rule) allows the use of H instances equivalent to $C\sigma$. Summing up, the resulting system is sound, but not refutationally sound.

Improvements over the Inference System

New Induction Schemas In the standard implementation, Generate produces intermediate instances of the current conjecture C using cover substitutions that instantiate some of its (induction) variables with elements of the cover set of their types. Therefore, the number of cover substitutions is exponential w.r.t. number of induction variables. For real applications such as the cross-validation of the JCVMs, this induction scheme may quite often generate thousands of cover substitutions. Such cases imply prover performances unacceptably poor.

In order to overcome this problem, we have implemented the following narrowing-based induction scheme, which leads to a major improvement in terms of performance: assume that there exists a (sub)term t of C whose head symbol is defined and that unifies with left-hand sides of the axioms defining the head symbol. From the most general unifier, we can immediately deduce that the cover substitution σ and the axiom that can simplify $C\sigma$. Therefore, the number of cover substitutions is limited to the number of axioms defining the head symbol, usually ranged to tens. As explained above, the improved schema is sound because the specification is constructor-based, complete and strongly complete. The last property guarantees that the disjunction of the conditions related to instances of axioms having the same left-hand side is valid.

Although of no incidence for our purpose, this scheme is not fit to prove conjectures having different (sub)terms that share induction variables, as for the associativity of the addition over naturals; it leads to a proof divergence.

AUTO SIMPLIFICATION: $(E \cup \{C[\overline{s = t}]\}, H) \xrightarrow[Ax]{spike} (E \cup \{C'\}, H)$

if $s >_e t$ and C' is the clause C rewritten with $s \to t$, excepting the term s of $s = t$

CONGRUENCE CLOSURE: $(E \cup \{\ldots s = t \ldots \wedge t = u \wedge \ldots \Rightarrow l\}, H) \xrightarrow[Ax]{spike} (E \cup \{C'\}, H)$

where $C' \equiv \ldots s = t \ldots \wedge t = u \wedge \ldots \mathbf{s} = \mathbf{u} \Rightarrow l$

AUGMENTATION: $(E \cup \{\underbrace{cond \Rightarrow l}_{C}\}, H) \xrightarrow[Ax]{spike} (E \cup \{cond \wedge t \Rightarrow l\}, H)$

if there exists a clause $cond' \Rightarrow p$ of $R \cup H_{\preceq_c C} \cup E_{\prec_c C}$ s.t. every literal of $cond'$ is subsumed by $cond$

Figure 5.3: New inference rules.

Therefore, we have adapted the following heuristics: recursively, if the (sub)term t shares induction variables with other (sub)term t' of C, compute also the cover substitutions and apply the heuristics for t' as for t. Since the number of (sub)terms of C is finite, this heuristics terminates. At the end, by the combination of the partial cover substitutions, it returns a set of cover substitutions such that the resulted instances of C can be simplified at any position corresponding to the terms treated by the heuristics (like t and t'). In our proofs, the number of cover substitutions still remains ranged to tens.

New Inference Rules The following inference rules, illustrated in Figure 5.3, have been added in order to exploit the conditions of conjectures:

1. **auto simplification**. It allows the rewriting of a conjecture with its negative literals, and allows to eliminate an existential variable from the rest of a conjecture as soon as it appears in a top position in equalities. Note that the order $>_e$ is an extension of the usual recursive path ordering to existential variables: for example, an existential variable x is always greater than any term that does not contain x and is not itself an existential variable;

2. **congruence closure**. If a conjecture contains as conditions the literals of the form $s = t$ and $t = u$ then the new literal $s = u$ is added to the conditions. The new literals are built using a completion algorithm having as input all the negative literals of the current conjecture. The procedure is refined by looking in priority for new equalities between constructor terms. If the head symbols of the both sides of a new equality are the same, we can derive new equalities by a decomposition operation, otherwise the clause is eliminated from the set of conjectures;

3. **augmentation**. Given a conditional clause, its conclusion can be added to the conditions *cond* of a conjecture if the conditions of the clause are discharged by *cond* [Boyer and Moore, 1988b]. The typical use of this rule in our applications is when the clause is a non-orientable user-defined lemma.

Additional applicability conditions are put such that each of these inference rules is an instance of the SIMPLIFY rule. Hence the soundness of these rules follows from the soundness of the abstract inference system A from Chapter 4.

The application strategy of the inference rules is standard, by trying firstly the SIMPLIFY rules that do not add new conjectures, then the other SIMPLIFY rules and, finally, the GENERATE rules.

Implementation Optimisations Another major improvement in terms of execution time is the recording of the failures of the inference rules application in order to avoid useless computation. Some of the recordings are performed at the level of clauses (for example, for subsumption), others at the level of terms (for example, for rewriting). If a rewrite operation with an unconditional rule fails at a given position of a conjecture, the rule's identification number is associated to that position such that the rule is avoided in the further rewriting operations as long as the term containing the position does not change.

5.3 Applications to JavaCard

In this section, we describe the results of our experiments of using the extended version of SPIKE to prove CDO, CDA and MON. For each instruction, we have three modules, one for each property CDA, CDO, and MON; this separation has no other purpose than convenience for carrying our experiments and collecting statistics. Each module consists of three parts: an algebraic specification, in our case parts of the description of the JCVMs, a logical theory to be proven, in our case some assumptions about the program and statements of CDA, CDO and MON, and a strategy that determines the prover's behavior during the proofs. The modules are available at https://members.loria.fr/SStratulat/files/verificard.tgz for the Linux distributions.

The module begins with the name of the specification, and follows by declaring the types (or sorts), constructor symbols, and defined functions of the specification. Then the behavior of defined functions is specified by means of clauses. The module is completed by fixing a proof strategy, and by declaring of the conjectures to be proven; note that conjectures are formulated as equational clauses.

5.3.1 CDO

The new version of SPIKE has been used to verify CDO for most instructions (41 out of 45). Figure 5.4 provides an excerpt of the SPIKE module used to prove CDO for the function CONV, that factors several of the conversion bytecodes of the JCVM: s2b (short to bytes), s2i (short to integers), i2b (integers to bytes), i2s (integers to short). Figure 5.1 provides some statistics about this experiment. In the second column, we indicate if the proof has been already done (yes) or not yet (n.y.). The third column presents the number of lemmas introduced by the user (and proved previously by SPIKE), while the other columns show respectively the number of Generate rules, normalization operations with unconditional rules, case rewriting with conditional rules, syntactic subsumption rules, tautology elimination rules and the execution time. Note that most proofs are automatic, i.e. do not require users to provide SPIKE with additional lemmas, and done in a reasonable time.

5.3.2 CDA and MON

We are now working on the proofs of CDA and MON, and have proven both properties for around half of the instructions. These properties are more challenging to prove than CDO, in particular because they rely on a number of invariants about JavaCard programs and the JCVM memory model. Thus users must provide appropriate invariants for the proofs to go through; as the formulation of such invariants can only be made during proofs, the benefits of automation are less clear for CDA and MON.

5.3.3 Assessment

We briefly comment on the effectiveness of the tool, and establish a comparison with our work on CertiCartes.

Automation SPIKE provides a reasonable level of automation, and there is no need to tune the strategy for each lemma to be proved. The best results are achieved with CDO, which for many bytecodes can be proved automatically, i.e. without requiring the users to provide intermediate lemmas. As explained above, the level of automation is lower for CDA and CDO. We detail below two directions for improvement.

Counterexamples SPIKE provides useful feedback to the specifier. By permitting the automatic refutation of false conjectures, SPIKE highlights, at a relatively low cost, possible problems in the specifications. This is essential for complex, large-scale formalisations which are bound to contain bugs, at least in their initial stages.

Expressivity SPIKE is expressive enough for specifying the virtual machines, and the properties CDO, CDA and MON. However it is not expressive enough to prove the correctness of the BCV, see below.

Comparison with CertiCartes CDA, CDO and MON have been proved independently in the Coq proof assistant. The comparison is without surprise, e.g. SPIKE provides a better level of automation than Coq, but on the other hand proofs that are not automatic may be harder to go through in SPIKE. Further, Coq is more expressive than SPIKE, and provides an adequate environment to specify (and prove) some properties of the JavaCard platform that cannot even be stated in SPIKE—most of the work reported in [Barthe and Dufay, 2003] cannot be cast in SPIKE.

Directions for improvement We see two directions of work for optimizing the usefulness of SPIKE in the context of the certification of the JavaCard Platform:

- **Automatic generation of intermediate lemmas:** for a number of bytecodes, SPIKE requires users to provide intermediate results for establishing CDO, CDA and MON. Many of such lemmas are of a similar

```
specification : CONV

sorts : list type_prim jcvm_state  ...

constructors :
Nil:  -> (list 'A ); Cons__:  'A (list 'A) -> (list 'A );...

defined functions :
CONV___ : type_prim type_prim jcvm_state -> returned_state;...

axioms :

% cONV :1

stack_f (u1) = Nil =>
cONV (u2, u3, u1) -> abortCode (State_error, u1) ;

% cONV :2

stack_f (u1) = Cons (e2, e3),
extr_from_opstack (u4, opstack(e2)) = Inl (Pair (e5, e6))
=> cONV (u4, u7, u1) -> update_frame
(push_opstack (VPrim (tpz2vp (u7, t_convert (u4, u7, e5))), e6, e2),
u1) ;

% cONV :3

stack_f (u1) = Cons (e2, e3),
extr_from_opstack (u4, opstack (e2)) = Inr (e5)
=> cONV (u4, u6, u1) -> abortCode (e5, u1) ;



 strategy : ...

 conjectures :

state = Build_jcvm_state( sh, hp, Cons (h, lf)),
res = cONV(n, z0, state) =>
res = abortCode( Type_error, state),
res = abortCode( Signature_error, state),
rs2ors( res) = ocONV (n, z0, s2os(state));
```

Figure 5.4: An excerpt of a SPIKE specification formalizing the instruction CONV.

instruction	proved	lemmas	Generate	U. R.	C. R.	Subsumption	Taut.	time
ACONST_NULL	yes	0	0	4	1	0	1	0.5s
ALOAD	n.y.	0	0	0	0	0	0	0.0
ARITH	yes	33	100	8771	2893	979	2178	8m
ARRAYLENGTH	yes	22	23	880	199	105	567	16s
ASTORE	n.y.	0	0	0	0	0	0	0.0
ATHROW	yes	24	24	2021	29	168	3496	1m42s
CHECKCAST	yes	79	88	1531	382	153	4741	1m44s
CONST	yes	0	1	24	7	0	7	0.5s
CONV	yes	0	94	999	405	12	495	0.54s
DUP	yes	0	4	21	3	2	26	0.13s
DUP2	yes	0	4	45	4	4	62	0.25s
GETFIELD	yes	24	49	4080	1074	347	4581	1m57s
GETFIELD_THIS	yes	24	49	4080	1074	347	4581	1m56s
GETSTATIC	yes	0	22	313	25	23	543	2.58s
GOTO	yes	0	0	4	1	0	1	0.07s
ICMP	yes	0	93	283	6	135	156	1.9s
IFNONNULL	yes	0	20	85	23	13	54	0.47s
IFNULL	yes	0	22	89	15	13	56	0.85s
IF_ACMP_COND	yes	13	38	147	31	48	99	1.3s
IF_COND	yes	0	46	175	117	97	81	0.4s
IF_SCMP_COND	yes	0	75	288	130	116	163	1.3s
INC	yes	0	10	217	29	22	566	1.5s
INSTANCEOF	yes	66	72	4173	1838	1203	5388	297m
INVOKEINTERFACE	yes	47	53	951	171	261	2026	0.38s
INVOKESPECIAL	yes	41	54	633	103	166	1097	13s
INVOKESTATIC	yes	8	12	42	7	13	72	0.4s
INVOKEVIRTUAL	yes	49	57	891	172	251	1576	31s
JSR	yes	0	0	4	1	0	1	0.15s
LOAD	yes	0	17	196	25	20	417	1.7s
LOOKUPSWITCH	yes	19	33	3434	1208	372	7414	52.4s
NEG	yes	2	16	87	38	24	76	1.2s
NEW	yes	1	7	26	3	4	33	0.17s
NEWARRAY	yes	22	31	4239	435	574	7540	1m07s
NOP	yes	0	0	4	1	0	1	0.05s
POP	yes	0	6	25	3	3	26	0.1s
POP2	yes	0	9	31	3	3	27	0.1s
PUSH	yes	0	1	9	1	0	10	0.4s
PUTFIELD	n.y.	0	0	0	0	0	0	0.0
PUTFIELD_THIS	n.y.	0	0	0	0	0	0	0.0
PUTSTATIC	yes	21	57	643	155	124	1096	9s
RET	yes	0	6	36	3	4	37	0.14s
RETURN	yes	8	11	200	33	2	199	0.92s
STORE	yes	0	55	554	139	95	2028	9.8s
SWAP	yes	0	13	51	4	5	64	0.3s
TABLESWITCH	yes	17	86	13651	10830	1190	4302	15m43

Table 5.1: Statistics for CDO proofs carried on a PC equipped with a 3.06 GHz Pentium processor and 512 Mbytes RAM.

shape, and could be generated automatically so as to minimize user interactions. One possibility that we are exploring is to exploit the abstraction script used to generate the offensive/abstract machine from the defensive one for generating such lemmas;

- **Connection with Coq:** while SPIKE provides a reasonable level of automation, not all proofs can be performed automatically. Further, Coq is more expressive than SPIKE, as explained above. In this respect, it may be beneficial to connect Coq and SPIKE so that Coq users may appeal to SPIKE to discharge some proof obligations automatically, as is done for example for Coq and Elan [Nguyen *et al.*, 2002]. In particular, such a connection would allow users to discharge automatically many trivial (sub)cases of CDA and MON, namely those which do not rely on any invariant. Some preliminary work is presented in Chapter 6.

5.4 Conclusions and related works

Our work is an attempt to apply rewriting techniques to validate security architectures for low-level languages used in smartcards. There have been a number of other applications of rewriting techniques to security, but these works focus on different aspects of security, such as network security and cryptographic protocols, see e.g. [Cervesato *et al.*, 1999, Denker *et al.*, 2000].

Our work is related to existing efforts to provide formal specification and correctness proofs for open platforms, including those carried by E. Giménez and co-authors at Trusted Logic, by J.-L. Lanet and co-workers at Gemplus [Casset *et al.*, 2002] (abstract B machines for the JCVM and BCV), by T. Nipkow and co-workers at Munich [Klein and Nipkow, 2003] (Java, JVM, BCV, and compiler in Isabelle), by J Strother Moore and co-workers at U. of Texas (JVM and BCV in ACL2)—note that some other works, e.g. [Stärk *et al.*, 2001, Coglio *et al.*, 1998] provide machine executable semantics, but pencil-and-paper formal proofs. We refer to [Hartel and Moreau, 2001] for further information, and limit ourselves to notice that most of these specifications implicitly use a restricted framework, but our work is distinctive by expliciting and taking advantage of this restricted framework.

Closest to our work is the work by A. Gordon and D. Syme [Syme and Gordon, 2002], which aims at automatic type-safety proofs for low-level languages. They identify a restricted framework in which specifications and properties can be expressed, and enhance the proof assistant HOL with suitable decision procedures, inspired from SVC (Stanford Validity Checker), to achieve a high degree of automation. Our methodology, which aims at validating automatically derived abstractions, seems crisper but we lack grounds for comparison —it would be interesting to validate, as they do, the .NET virtual machine, for carrying such a comparison.

6

Certifying Formula-based Noetherian Induction Reasoning

Sommaire

6.1 Formalising formula-based Noetherian induction proofs 64
6.1.1 Formalising the induction ordering and measure values with COCCINELLE 64
6.1.2 Proving formulas from LF 66
6.2 Certifying implicit induction proofs 67
6.2.1 Inference systems for implicit induction 67
6.2.2 A first attempt to certify the proof of $q(x, y) = true$ 68
6.2.3 Coq formalizations of implicit induction proofs 69
6.2.4 Certification of implicit induction proofs for Coq formalisations using the functional programming style 72
6.3 Certifying cyclic proofs 74
6.3.1 Inference systems for cyclic induction 74
6.3.2 Cyclic proofs 76
6.3.3 Certification of cyclic proofs for Coq formalisations using the logic programming style 79
6.3.4 Certification of cyclic proofs for Coq formalisations using the functional programming style 80
6.4 Automatic certification of SPIKE proofs 84
6.5 Conclusions 84

Compared to the term-based Noetherian induction instances, the formula-based instances are not directly supported by the current proof assistants. In this chapter, we present 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 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.

The content of the chapter is based on the papers [Stratulat, 2010, Stratulat and Demange, 2011, Henaien and Stratulat, 2013, Stratulat, 2017b].

Structure of the chapter. The chapter is organised in five sections, as follows. The Coq formalisation of the formula-based Noetherian induction principle and the certification methodology are presented in Section 6.1. Section 6.2 introduces different implicit induction inference systems. A first attempt to certify the proofs of the theorems for the P&Q example, given in the General Introduction, is presented. Afterwards, 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 6.3 using cyclic inference systems and showed that the certification of the cyclic proof concerning the P&Q example is successful by using a logic programming style. The instructions for generating implicit and cyclic proofs by the SPIKE prover as well as their conversion into Coq script are presented in Section 6.4. Section 6.5 concludes.

The full source code, including the used libraries, reasoning systems, specifications and proofs, is provided as supplementary material at https://members.loria.fr/SStratulat/files/jsc-pas.zip and on SPIKE's website https://github.com/sorinica/spike-prover.

6.1 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* 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 $<_f$ as a multiset extension of an 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 6.2, 6.3, and 6.4.

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, Contejean *et al.*, 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.

6.1.1 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 the rpo, denoted by **rpo**, and its multiset extension **rpo_mul**, 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&Q example, they (and other intermediary functions) are defined as:

```
Inductive symb : Set :=
  | id_zero
  | id_succ
  | id_P
  | id_Q

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

Definition status (f:symb) :=
  match f with
    | id_zero ⇒ Mul
    | id_succ ⇒ Mul
    | id_P ⇒ Mul
    | id_Q ⇒ Mul
  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 type employed in the specification, a new *model* function translating its constructor terms can be defined. For example, the translation function defined for **nat**, the Coq builtin type for naturals built from the constructors O and S, is:

  ```
  Fixpoint model_n (v: nat): term :=
    match v with
      | O ⇒ (Term id_0 nil)
      | (S x) ⇒ let r := model_n x in
                    (Term id_S (r::nil))
    end.
  ```

  For non-inductive types, we have to define translation axioms. E.g., the translation axioms for the type T from the P&Q example, are:

  ```
  Axiom model_nat_0: model_nat zero = (Term id_zero nil).
  Axiom model_nat_succ: ∀ (u1: T), model_nat (succ u1) = (Term id_succ ((model_nat u1)::nil)).
  ```

- 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 given as a Coq list;
- the COCCINELLE counterpart of any variable x of sort s will be the term (model_s x).

Example 12 *The measure value used for (**Q** x zero) in the formula-based Noetherian induction proof from Figure 3 can be represented as the following COCCINELLE term list: (model_nat x :: model_nat x :: (Term id_zero nil) :: nil), corresponding to the literate translation of the multiset* $\{x, x, zero\}$.

6.1.2 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 : $\forall$ F, In F LF $\rightarrow$ $\forall$ u, ($\forall$ F', In F' LF $\rightarrow$ $\forall$ u', less (snd (F' u')) (snd (F u)) $\rightarrow$ fst (F' u')) $\rightarrow$ 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: $\forall$ F, In F LF $\rightarrow$ $\forall$ u: **nat**, fst (F u).

As a case study, the methodology will be used in Sections 6.2 and 6.3 for certifying proofs of conjectures about conditional specifications.

6.2 Certifying implicit induction proofs

6.2.1 Inference systems for implicit induction

We define a minimalistic and concrete implicit induction inference system, denoted by I_s^b, able to prove the conjectures from the P&Q example.

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

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

RedEq (R): $(E \cup \{\phi\},\ H) \vdash_{I_s^b} (E \cup \bigcup_i \{l_i\sigma = r_i\sigma\},\ H \cup \{\phi\})$,
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 specification of the predicates P and Q will be done using conditional axioms, built from the translation of the inductive definitions of the predicates P and Q into the boolean functions p and q, respectively:

$$p(0) = true \tag{6.1}$$
$$p(x) = true \wedge q(x, S(x)) = true \Rightarrow p(S(x)) = true \tag{6.2}$$
$$p(x) = false \Rightarrow p(S(x)) = false \tag{6.3}$$
$$q(x, S(x)) = false \Rightarrow p(S(x)) = false \tag{6.4}$$

$$q(x, 0) = true \tag{6.5}$$
$$p(x) = true \wedge q(x, y) = true \Rightarrow q(x, S(y)) = true \tag{6.6}$$
$$p(x) = false \Rightarrow q(x, S(y)) = false \tag{6.7}$$
$$q(x, y) = false \Rightarrow q(x, S(y)) = false \tag{6.8}$$

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

$(\{\underline{q(x,y) = true}\}, \emptyset)$ $\vdash^I_{I_s^b}$
$(\{\underline{q(x,0) = true}, q(x, S(y')) = true\}, \emptyset)$ $\vdash^D_{I_s^b}$
$(\{\underline{q(x,S(y')) = true}\}, \emptyset)$ $\vdash^R_{I_s^b}$
$(\{\underline{p(x) = true}, q(x,y') = true\}, \{q(x, S(y')) = true\})$ $\vdash^I_{I_s^b}$
$(\{\underline{p(0) = true}, p(S(x')) = true, q(x,y') = true\}, \{q(x, S(y')) = true\})$ $\vdash^D_{I_s^b}$
$(\{\underline{p(S(x')) = true}, q(x,y') = true\}, \{q(x, S(y')) = true\})$ $\vdash^R_{I_s^b}$
$(\{p(x') = true, \underline{q(x', S(x')) = true}, q(x,y') = true\},$
$\{p(S(x')) = true, q(x, S(y')) = true\}) \vdash^D_{I_s^b}$
$(\{\underline{p(x') = true}, q(x,y') = true\}, \{p(S(x')) = true, q(x, S(y')) = true\})$ $\vdash^I_{I_s^b}$
$(\{\underline{p(0) = true}, p(S(x'')) = true, q(x,y') = true\},$
$\{p(S(x')) = true, q(x, S(y')) = true\}) \vdash^D_{I_s^b}$
$(\{\underline{p(S(x'')) = true}, q(x,y') = true\}, \{p(S(x')) = true, q(x, S(y')) = true\})$ $\vdash^D_{I_s^b}$
$(\{\underline{q(x,y') = true}\}, \{p(S(x')) = true, q(x, S(y')) = true\})$ $\vdash^I_{I_s^b}$
$(\{\underline{q(x,0) = true}, q(x, S(y'')) = true\}, \{p(S(x')) = true, q(x, S(y')) = true\})$ $\vdash^D_{I_s^b}$
$(\{\underline{q(x,S(y'')) = true}\}, \{p(S(x')) = true, q(x, S(y')) = true\})$ $\vdash^D_{I_s^b}$
$(\emptyset, \{p(S(x')) = true, q(x, S(y')) = true\})$

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

Theorem 12 (soundness of I_s^b) *The inference system I_s^b is sound.*

Proof It follows directly from the soundness of the inference system A, presented in Section 4.1.2, if we prove that each I_s^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. This is 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 12, we can conclude that $q(x, y) = true$ is a consequence of the axioms defining p and q.

6.2.2 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, Coq 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 6.3.3 how to handle this situation using the logic programming style.

6.2.3 Coq formalizations of implicit induction proofs

In the rest of the section, we will stick to formalize 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:

$$even(0) = true \quad (6.9) \qquad odd(0) = false \quad (6.11)$$
$$even(S(x)) = odd(x) \quad (6.10) \qquad odd(S(x)) = even(x) \quad (6.12)$$

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

$$0 + x = x \quad (6.13) \qquad S(x) + y = S(x + y) \quad (6.14)$$

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

$$odd(u_1 + u_2) = true \wedge even(u_2 + u_3) = true \Rightarrow odd(u_1 + u_3) = true \quad (6.15)$$

by using reductive reasoning techniques. *Rewriting* is a most effective reductive technique for reasoning on equational specifications.

Let $\prec$ be a reduction ordering over terms and ρ a set 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 13 *The axioms (6.9)-(6.14) can be oriented from left to right using as reduction ordering the mpo 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 13 (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 (6.15) can be built using the inference system I_s^f:

GenNat (G): $(E \cup \{\phi\langle x\rangle\},\ H) \vdash_{I_s^f} (E \cup \{\phi_1, \phi_2\},\ H \cup \{\phi\})$,
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): $(E \cup \{\phi\},\ H) \vdash_{I_s^f} (E \cup \Phi,\ H)$,
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): $(E \cup \{\phi\}, H) \vdash_{I_s^f} (E,\ H)$,
if ϕ is an instance of an equality from H.

Negative Clash (N): $(E \cup \{\phi\}, H) \vdash_{I_s^f} (E,\ H)$,
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_s^f-proof of (6.15) 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:

$(\{\underline{e_13(u_1, u_2, u_3)}\}, \emptyset)$ $\vdash^{G}_{I^{c}_{f}}$
$(\{\underline{e_23(u_2, u_3), e_29(u_4, u_2, u_3)}\}, \{e_13(u_1, u_2, u_3)\})$ $\vdash^{*S}_{I^{f}_{c}}$
$(\{\underline{e_36(u_2, u_3)}, e_49(u_4, u_2, u_3)\}, \{e_13(u_1, u_2, u_3)\})$ $\vdash^{G}_{I^{c}_{f}}$
$(\{\underline{e_67(u_3), e_73(u_4, u_3)}, e_49(u_4, u_2, u_3)\}, \{e_36(u_2, u_3), e_13(u_1, u_2, u_3)\})$ $\vdash^{*S}_{I^{f}_{s}}$
$(\{\underline{e_81(u_3)}, e_91(u_4, u_3), e_49(u_4, u_2, u_3)\}, \{e_36(u_2, u_3), e_13(u_1, u_2, u_3)\})$ $\vdash^{N}_{I^{f}_{s}}$
$(\{\underline{e_73(u_4, u_3)}, e_49(u_4, u_2, u_3)\}, \{e_36(u_2, u_3), e_13(u_1, u_2, u_3)\})$ $\vdash^{S}_{I^{f}_{s}}$
$(\{e_91(u_4, u_3), \underline{e_49(u_4, u_2, u_3)}\}, \{e_36(u_2, u_3), e_13(u_1, u_2, u_3)\})$ $\vdash^{G}_{I^{f}_{s}}$
$(\{e_91(u_4, u_3), \underline{e_113(u_2, u_3), e_119(u_5, u_2, u_3)}\}, \{e_49(u_4, u_2, u_3), e_36(u_2, u_3), e_13(u_1, u_2, u_3)\})$ $\vdash^{*S}_{I^{f}_{s}}$
$(\{e_91(u_4, u_3), e_138(u_2, u_3), \underline{e_13(u_5, u_2, u_3)}\}, \{e_49(u_4, u_2, u_3), e_36(u_2, u_3), e_13(u_1, u_2, u_3)\})$ $\vdash^{E}_{I^{f}_{s}}$
$(\{\underline{e_91(u_4, u_3)}, e_138(u_2, u_3)\}, \{e_49(u_4, u_2, u_3), e_36(u_2, u_3), e_13(u_1, u_2, u_3)\})$ $\vdash^{G}_{I^{f}_{s}}$
$(\{\underline{e_189(u_3)}, e_195(u_5, u_3), e_138(u_2, u_3)\}, \{e_91(u_4, u_3), e_49(u_4, u_2, u_3), e_36(u_2, u_3), e_13(u_1, u_2, u_3)\})$ $\vdash^{S}_{I^{f}_{s}}$
$(\{\underline{e_195(u_5, u_3)}, e_138(u_2, u_3)\}, \{e_91(u_4, u_3), e_49(u_4, u_2, u_3), e_36(u_2, u_3), e_13(u_1, u_2, u_3)\})$ $\vdash^{*S}_{I^{f}_{s}}$
$(\{\underline{e_36(u_5, u_3)}, e_138(u_2, u_3)\}, \{e_91(u_4, u_3), e_49(u_4, u_2, u_3), e_36(u_2, u_3), e_13(u_1, u_2, u_3)\})$ $\vdash^{E}_{I^{f}_{s}}$
$(\{\underline{e_138(u_2, u_3)}\}, \{e_91(u_4, u_3), e_49(u_4, u_2, u_3), e_36(u_2, u_3), e_13(u_1, u_2, u_3)\})$ $\vdash^{G}_{I^{f}_{s}}$
$(\{\underline{e_237(u_3)}, e_243(u_5, u_3)\}, \{e_138(u_2, u_3), e_91(u_4, u_3), e_49(u_4, u_2, u_3), e_36(u_2, u_3), e_13(u_1, u_2, u_3)\})$ $\vdash^{S}_{I^{f}_{s}}$
$(\{\underline{e_243(u_5, u_3)}\}, \{e_138(u_2, u_3), e_91(u_4, u_3), e_49(u_4, u_2, u_3), e_36(u_2, u_3), e_13(u_1, u_2, u_3)\})$ $\vdash^{*S}_{I^{f}_{s}}$
$(\{\underline{e_267(u_5, u_3)}\}, \{e_138(u_2, u_3), e_91(u_4, u_3), e_49(u_4, u_2, u_3), e_36(u_2, u_3), e_13(u_1, u_2, u_3)\})$ $\vdash^{G}_{I^{f}_{s}}$
$(\{\underline{e_295(u_3), e_301(u_6, u_3)}\}, \{e_267(u_5, u_3), e_138(u_2, u_3), e_91(u_4, u_3), e_49(u_4, u_2, u_3), e_36(u_2, u_3), e_13(u_1, u_2, u_3)\})$ $\vdash^{*S}_{I^{f}_{s}}$
$(\{\underline{e_317(u_3)}, e_138(u_6, u_3)\}, \{e_267(u_5, u_3), e_138(u_2, u_3), e_91(u_4, u_3), e_49(u_4, u_2, u_3), e_36(u_2, u_3), e_13(u_1, u_2, u_3)\})$ $\vdash^{N}_{I^{f}_{s}}$
$(\{\underline{e_138(u_6, u_3)}\}, \{e_267(u_5, u_3), e_138(u_2, u_3), e_91(u_4, u_3), e_49(u_4, u_2, u_3), e_36(u_2, u_3), e_13(u_1, u_2, u_3)\})$ $\vdash^{E}_{I^{f}_{s}}$
$(\emptyset, \{e_267(u_5, u_3), e_138(u_2, u_3), e_91(u_4, u_3), e_49(u_4, u_2, u_3), e_36(u_2, u_3), e_13(u_1, u_2, u_3)\})$

As for the I^{b}_{s}-proof of the equality $q(x, y) = true$, the processed conjectures from each state are underlined. $\vdash^{*S}_{I^{f}_{s}}$ 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 14 (soundness of I^{f}_{s}) *The inference system I^{f}_{s} is sound.*

Proof As for the proof of Theorem 12, we show that each I^{f}_{s}-rule is the instance of an A-rule.

- GenNat is an instance of the two-step version of AddPremise, presented in Figure 8.2 and explained in Subsection 8.1.2. 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 14, we conclude that the conjecture (6.15) is a consequence of the axioms defining *even*, *odd* and '+'.

6.2.4 Certification of implicit induction proofs for Coq formalisations using the functional programming style

The induction ordering is built similarly as for the P&Q example, shown in Subsection 6.1.1, 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 u1, (f u1). 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 resulting from the rewriting of ϕ in the I_s^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 13, 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.
```

6.3 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 (see Chapter 5). 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 Chapter 2 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 6.2, the validation of cyclic proofs needs fewer ordering constraints and shorter LF lists.

6.3.1 Inference systems for cyclic induction

We introduce the abstract inference system D', similar to the system D from Chapter 2, which consists of the following three rules:

DEDUCTION: $E \cup \{\phi\} \vdash_{D'} E \cup \Phi$, where
i) $\forall \psi \in \Phi, Var(\psi) \subseteq Var(\phi)$, and
ii) Φ has a counterexample whenever ϕ has a counterexample.

SPLIT: $E \cup \{\phi\} \vdash_{D'} E \cup \Phi$, where
Φ is $\bigcup\{\phi\sigma \mid \sigma \equiv \bigcup_i \{x_i \mapsto t_i\}$ and $\forall i, x_i \in Var(\phi)$ and $Var(t_i)$ are fresh$\}$, and
whenever ϕ has a counterexample $\phi\tau$, Φ has $\phi\tau$ as counterexample.

INDUCTION: $E \cup \{\phi\} \vdash_{D'} E \cup \Phi$, where
i) there exists an instance $\psi\delta$ of a previously generated equality ψ such that
Φ or $\psi\delta$ has a counterexample whenever ϕ has a counterexample, and
ii) $Var(\psi\delta) \subseteq Var(\phi)$ and $\forall \psi' \in \Phi, Var(\psi') \subseteq Var(\phi)$.

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 3 ($D' - preproof$) *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_s^b from Section 6.2, 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 15 *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 14 *The following I_c^b-preproof of $\{q(x,y) = true\}$ can be built by using the axioms (6.1)-(6.8) and the proof scenario given at the general introduction:*

$\{\underline{q(x,y) = true}\} \vdash_{I_c^b}^{S_c} \{\underline{q(x,0) = true}, q(x,S(y')) = true\} \vdash_{I_c^b}^{D_c} \{\underline{q(x,S(y')) = true}\} \vdash_{I_c^b}^{R_c} \{\underline{q(x,y') = true}, p(x) = true\} \vdash_{I_c^b}^{I_c} \{\underline{p(x) = true}\} \vdash_{I_c^b}^{S_c} \{\underline{p(0) = true}, p(S(x')) = true\} \vdash_{I_c^b}^{D_c} \{\underline{p(S(x')) = true}\} \vdash_{I_c^b}^{R_c} \{\underline{p(x') = true}, q(x',S(x')) = true\} \vdash_{I_c^b}^{I_c} \{\underline{q(x',S(x')) = true}\} \vdash_{I_c^b}^{I_c} \emptyset$,
where the processed conjectures are underlined.

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 15 *The I_c^b-preproof from Example 14 is illustrated as the oriented graph from Figure 6.1.*

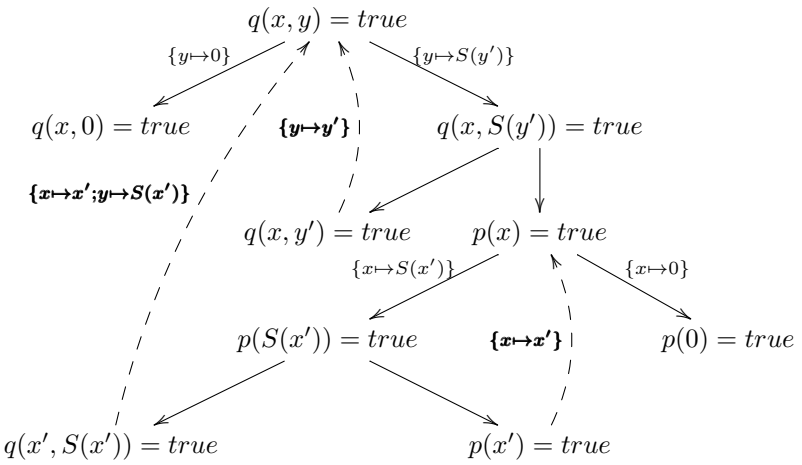


Figure 6.1: The I_c^b-preproof of $\{q(x,y) = true\}$ as an oriented graph.

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.

Example 16 *The oriented graph from Figure 6.1 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*
- $[q(x,y) = true, q(x,S(y') = true, p(x) = true, p(S(x')) = true, q(x',S(x')) = true]$

By abuse of notation, the nodes from the paths have been denoted by the labelling equalities.

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.

6.3.2 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 p, we define ordering constraints 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 Figure 6.2 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_{id}^{ϕ}, 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.

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 17 *The I_c^b-preproof from Figure 6.1 has one non-root IH-node, labelled by $p(x) = true$. The result of the transformation applied on it is illustrated in Figure 6.3. The minimal cycles of the strongly connected component are all normalised:*

- *first cycle:* $[q(x,u) = true, q(x,S(y')) = true, q(x,y') = true]$,

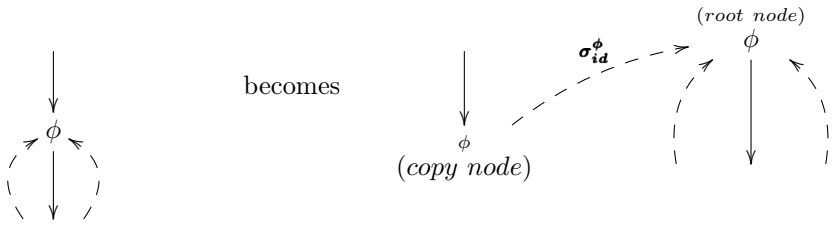


Figure 6.2: The transformation of a non-root IH-node.

- *second cycle:* $[p(x) = true, p(S(x')) = true, p(x') = true]$, *and*
- *third cycle:* $[q(x,y) = true, q(x, S(y')) = true, p(x) = true]$, $[p(x) = true, p(S(x')) = true, q(x', S(x')) = true]$.

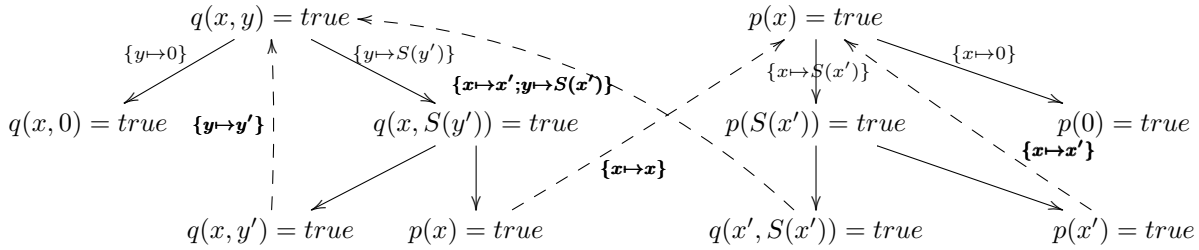


Figure 6.3: The normalised I^b_c-preproof of $\{q(x,y) = true\}$.

Lemma 6 *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 18 *Figure 6.3 also is a I^b_c-preproof for $\{p(x) = true, q(x,y) = true\}$.*

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 $\sigma_1 \cdots \sigma_k$, where each σ_i ($i \in [1..k]$) is the substitution associated to the node n_i. $\phi(n)$ denotes the equality labelling the node n.

Definition 4 (n-cycle discharging IHs) *An n-cycle, made of a circular list of $n(>0)$ paths $[n^1_1, \ldots n^{p1}_1], \ldots, [n^1_n, \ldots, n^{p_n}_n]$ from a strongly connected component p,* discharges *the IHs $\phi(n^1_j)\delta_j$ ($j \in [1..n]$) if, for any $i \in [1..n]$, we have that $\phi(n^1_{next(i)})\delta_{next(i)} <_p \phi(n^1_i)\theta_i$, where θ_i is the cumulative substitution for the path $[\phi(n^1_i), \ldots, \phi(n^{p_i}_i)]$, $next(i) = 1 + (i \bmod n)$ and $<_p$ is a well-founded and 'stable under substitutions' ordering defined over the instances of the equalities labelling the root nodes of p.*

Example 19 *The IHs used in the minimal cycles from Figure 6.3 and detailed in Example 17 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_c^b-preproof in Figure 6.3. 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 6.2, 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 mpo based on the precedence $false <_\mathcal{F} true <_\mathcal{F} 0 <_\mathcal{F} s$.

Theorem 16 *[soundness of D'-preproofs] Any D'-preproof is sound if the minimal cycles from its normal form discharge their IHs.*

Proof Let $\mathcal{P}$ be a D'-preproof and $\mathcal{P}'$ its normal form such that all minimal cycles from $\mathcal{P}'$ discharge their IHs. By contradiction, we assume that $\mathcal{P}$ is not sound, i.e., there is an equality e proved by $\mathcal{P}$ that is false. By Lemma 6, e labels one of the root nodes of $\mathcal{P}'$.

The set $\mathcal{C}$ of strongly connected components from the graph of $\mathcal{P}'$ forms a partition of the nodes from the graph of $\mathcal{P}'$. Let $<_\mathcal{C}$ denote the well-founded ordering defined over the elements of $\mathcal{C}$. We perform a classical induction reasoning over the elements of $\mathcal{C}$.

The base case. Let us assume that e labels the root node n from one of the $<_\mathcal{C}$-minimal strongly connected components of $\mathcal{C}$, denoted by p. Let us also assume that n is the only node from p. Hence, $\phi(n) \equiv e$. Since $\phi(n)$ is false, it cannot label a leaf node, so there is a D'-rule that was applied on $\phi(n)$. The nodes labeled by the resulting equalities, as well as the corresponding IH-node for the case when Induction was applied on $\phi(n)$, should be from p, by the minimality of p. This contradicts the fact that p has only one node.

Therefore, p should have at least two nodes. Again, let n be some root node from p such that $\phi(n) \equiv e$. We will show that Induction should be applied with a false IH on $\phi(n)$ or on an equality deriving from $\phi(n)$. We will perform a case analysis:

- We assume that Deduction or Split were applied on $\phi(n)$ in the inference step $E \cup \{\phi(n)\} \vdash_{D'} E \cup \Phi$. Since $\phi(n)$ is false, it has a counterexample. By the definition of these rules, there is an equality in Φ that has a counterexample.
- If Induction was applied on $\phi(n)$ in the inference step $E \cup \{\phi(n)\} \vdash_{D'} E \cup \Phi$, using an instance of the previously generated equality ψ as IH, since $\phi(n)$ is false, it has a counterexample. Cf. Induction definition, either Φ or ψ has a counterexample.

Any rule applied on $\phi(n)$ can also be applied on some of its counterexamples $\phi(n)\tau$ for which the IH and new equalities are ground because the IH and new equalities have variables included in $Var(\phi(n))$. When a rule is applied on $\phi(n)\tau$, we can have that either i) there is an offspring of n, denoted by n', such that $\phi(n')\tau$ is a counterexample, or ii) Induction was applied on $\phi(n)$ using the IH h and $h\tau$ is a counterexample. For the case ii), we are done, as required. For the case i), we can apply the same reasoning on n' as for n. Either Induction was applied with a false IH and we are done, or again case i) holds. This process cannot continue forever since the visited nodes follow some path in the tree derivation rooted by n, which is finite, so case ii) eventually happens.

We denote by n_f the Induction-node identified by this process, by n_h its IH-node and by $\phi(n_h)\delta$ the false coresponding IH, knowing that n and n_f may refer to the same node. The path $[n, \ldots, n_f]$ is part of a cycle c from p, of the form $[n_h, \ldots], \ldots, [n, \ldots, n_f]$, that links back n_h to n. We will show that there is a minimal n-cycle c_m of the form $[n_h, \ldots], \ldots, [n, \ldots, n_f]$ which may be different from c. If c is minimal, then c_m will denote c. Otherwise, c includes another cycle c', meaning that c' has fewer nodes. In this case, the nodes of c can be described as the union of the nodes of another cycle c'', also with fewer nodes, and the nodes of c'. If $[n, \ldots, n_f]$ is a path from c', then we apply the same reasoning on c' as for c. Otherwise, $[n, \ldots, n_f]$ is a path from c'' and we can also apply the same reasoning on c'' as for c. This process should finish because the number of nodes in p is finite. We denote by c_m the last cycle including $[n, \ldots, n_f]$. Hence, c_m is minimal, of the form $[n_h, \ldots], \ldots, [n, \ldots, n_f]$. Then, the IH $\phi(n_h)\delta$ is discharged by c_m, and we have that $\phi(n)\theta >_p \phi(n_h)\delta$, where θ is the cumulative substitution for the path $[n, \ldots, n_f]$.

We have that $\phi(n)\tau, \ldots, \phi(n_f)\tau$ are the counterexamples identified during the process, and the IH $\phi(n_h)\delta\tau$ is a counterexample. It can be noticed that $\phi(n_f)$ can be generated from $\phi(n)\theta$ by applying only the Deduction rules used on the equalities labeling the nodes encountered when traversing the path $[n, \ldots, n_f]$. So, $\phi(n_f)\tau$ can be derived from $\phi(n)\theta\tau$ by applying the same rules. This is possible only if $\phi(n)\tau$ is an instance of $\phi(n)\theta$, i.e.,

there is a matching substitution σ such that $\phi(n)\theta\sigma \equiv \phi(n)\tau$. Since $\phi(n_f)$ can be generated from $\phi(n)\theta$, we have $\phi(n_f)\sigma \equiv \phi(n_f)\tau$. On the other hand, the substitution θ replaces a set $\overline{x}$ of variables from $\phi(n)$ by terms with fresh variables, so no variable from $\overline{x}$ is in $\phi(n_f)$. Since $\phi(n_f)\sigma \equiv \phi(n_f)\tau$, the terms mapped in σ and τ by the variables occurring in $\phi(n)$ should be the same. Therefore, we have $\phi(n)\theta\sigma \equiv \phi(n)\theta\tau$. So, $\phi(n)\theta\tau \equiv \phi(n)\tau$. By the 'stability under substitutions' property of $<_p$, we have that $\phi(n)\theta\tau >_p \phi(n_h)\delta\tau$, so $\phi(n)\tau >_p \phi(n_h)\delta\tau$.

We can apply a similar reasoning on $\phi(n_h)\delta\tau$ as for $\phi(n)\tau$ and show that there is a minimal n-cycle c'_m of the form $[n'_h, \ldots], \ldots, [n_h, \ldots, n'_f]$, for which $\phi(n'_h)\delta'_h$ is the IH used by the Induction rule applied on $\phi(n'_f)$. It can be shown that $\phi(n_h)\delta\tau >_p \phi(n'_h)\delta'_h\delta\tau$ holds. A similar reasoning can be done on the counterexample $\phi(n'_h)\delta'_h\delta\tau$ as for $\phi(n_h)\delta\tau$, and so on *ad infinitum*, to build the infinite strictly $<_p$-decreasing sequence

$$\phi(n)\tau >_p \phi(n_h)\delta\tau >_p \phi(n'_h)\delta'_h\delta\tau >_p \ldots$$

This contradicts the fact that $<_p$ is well-founded.

The step case. Let us assume that e labels the root node of an arbitrary non $<_{\mathcal{C}}$-minimal strongly connected component p from $\mathcal{C}$. By induction hypothesis, we assume that the root nodes of any strongly connected component $<_{\mathcal{C}}$-smaller than p are labelled by true equalities. If p has only one node, the only rule that can be applied on e is Induction. The new equalities and the IH should label root nodes from strongly connected components $<_{\mathcal{C}}$-smaller than p, hence they are true by induction hypothesis. By the definition of Induction, e should be true, hence contradiction.

So, p should have at least two nodes. By using similar arguments as for the base case and using the induction hypothesis, we can show that an infinite strictly $<_p$-decreasing sequence of counterexamples of equalities labeling root nodes from p can be built. □

Example 20 (cont. Example 19) *The I^b_c-preproof from Figure 6.3 is sound because all the IHs from its minimal cycles are discharged. Therefore, $q(x, y) = true$ and $p(x) = true$ are true.*

6.3.3 Certification of cyclic proofs for Coq formalisations using the logic programming style

Let us recall the definition of the inductive predicates **P** and **Q** from the P&Q example, by following the logic programming style:

```
Inductive P: T → Prop :=
  p1: P zero
| p2: ∀ x, (P x ∧ Q x (succ x)) → P (succ x)
with
  Q: T → T → Prop :=
  q1: ∀ y, Q y zero
| q2: ∀ x y, (Q x y ∧ P x) → Q x (succ 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 := T → T → 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 19:

```
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 : $\forall$ F, In F LF_PandQ $\rightarrow$ $\forall$ u, ($\forall$ F', In F' LF_PandQ $\rightarrow$ $\forall$ u', less (snd (F' u')) (snd (F u)) $\rightarrow$ fst (F' u')) $\rightarrow$ 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 6.2.

The proof of each resulted case follows some path from a root node to a leaf node in the cyclic graph from Figure 6.3. 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: $\forall$ F, In F LF_PandQ $\rightarrow$ $\forall$ u_1: T, fst (F u_1).

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

`Theorem` *true_Pu* : $\forall$ u, P u.

`Theorem` *true_Qxy* : $\forall$ x y, Q x y.

6.3.4 Certification of cyclic proofs for Coq formalisations using the functional programming style

Conjecture (6.15) can also be proved using the cyclic inference system I_c^f, based on the reasoning techniques underlying I_s^f, presented in Section 6.2:

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 17 *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 15;
- 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 15.

□

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 $\phi\{x \mapsto 0\} \to_{Ax} \phi_1$, $\phi\{x \mapsto S(x')\} \to_{Ax} \phi_2$ 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 (6.15), represented as a linear derivation:

$$\{\underline{e'_{13}(u_1,u_2,u_3)}\} \vdash^{G_c}_{I_c^f} \{\underline{e'_{23}(u_2,u_3)}, e'_{29}(u_4,u_2,u_3)\} \vdash^{A_c}_{I_c^f} \{e'_{36}(u_2,u_3), \underline{e'_{29}(u_4,u_2,u_3)}\} \vdash^{*A_c}_{I_c^f}$$
$$\{\underline{e'_{36}(u_2,u_3)}, e'_{49}(u_4,u_2,u_3)\} \vdash^{G_c}_{I_c^f} \{\underline{e'_{67}(u_3)}, e'_{73}(u_4,u_3), e'_{49}(u_4,u_2,u_3)\} \vdash^{A_c}_{I_c^f} \{\underline{e'_{81}(u_3)}, e'_{73}(u_4,u_3), e'_{49}(u_4,u_2,u_3)\} \vdash^{N_c}_{I_c^f}$$
$$\{\underline{e'_{73}(u_4,u_3)}, e'_{49}(u_4,u_2,u_3)\} \vdash^{*A_c}_{I_c^f} \{\underline{e'_{91}(u_4,u_3)}, e'_{49}(u_4,u_2,u_3)\} \vdash^{G_c}_{I_c^f} \{\underline{e'_{197}(u_3)}, e'_{203}(u_5,u_3), e'_{49}(u_4,u_2,u_3)\} \vdash^{T_c}_{I_c^f}$$
$$\{\underline{e'_{203}(u_5,u_3)}, e'_{49}(u_4,u_2,u_3)\} \vdash^{*A_c}_{I_c^f} \{\underline{e'_{36}(u_5,u_3)}, e'_{49}(u_4,u_2,u_3)\} \vdash^{E_c}_{I_c^f} \{\underline{e'_{49}(u_4,u_2,u_3)}\} \vdash^{G_c}_{I_c^f}$$
$$\{e'_{113}(u_2,u_3), \underline{e'_{119}(u_5,u_2,u_3)}\} \vdash^{*A_c}_{I_c^f} \{e'_{113}(u_2,u_3), \underline{e'_{13}(u_5,u_2,u_3)}\} \vdash^{E_c}_{I_c^f} \{\underline{e'_{113}(u_2,u_3)}\} \vdash^{A_c}_{I_c^f} \{\underline{e'_{138}(u_2,u_3)}\} \vdash^{G_c}_{I_c^f}$$

$\{\underline{e'_{253}(u_3)}, e'_{259}(u_5,u_3)\} \vdash^{T_c}_{I_c^f} \{\underline{e'_{259}(u_5,u_3)}\} \vdash^{*A_c}_{I_c^f} \{\underline{e'_{283}(u_5,u_3)}\} \vdash^{G_c}_{I_c^f} \{\underline{e'_{311}(u_3)}, e'_{317}(u_6,u_3)\} \vdash^{A_c}_{I_c^f}$
$\{\underline{e'_{333}(u_3)}, e'_{317}(u_6,u_3)\} \vdash^{N_c}_{I_c^f} \{\underline{e'_{317}(u_6,u_3)}\} \vdash^{*A_c}_{I_c^f} \{\underline{e'_{138}(u_6,u_3)}\} \vdash^{E_c}_{I_c^f} \emptyset$

Figure 6.4 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.

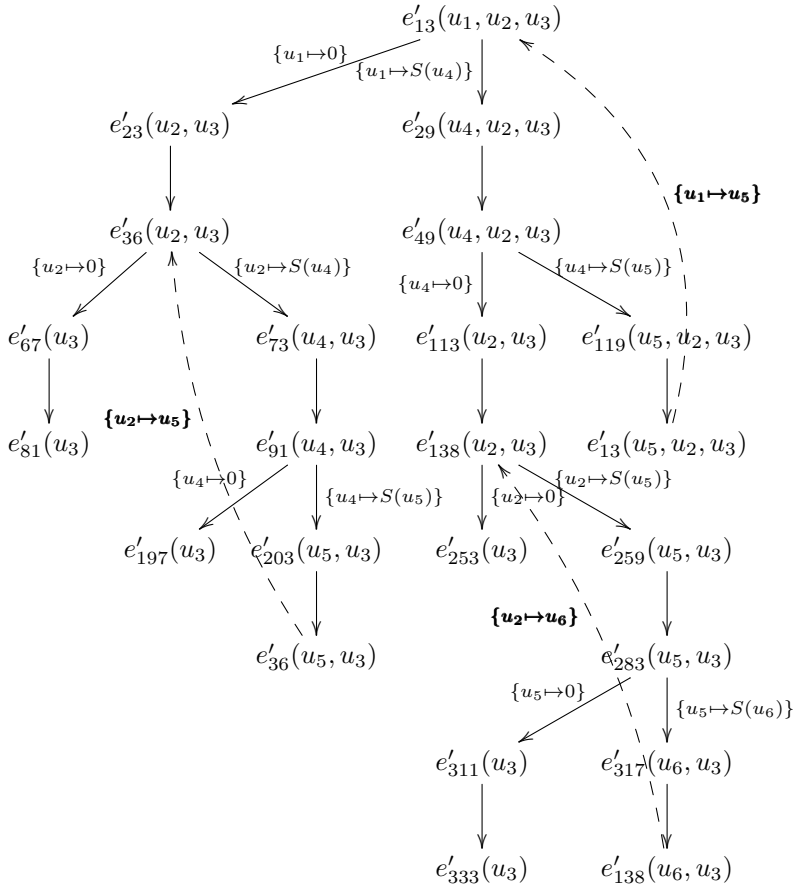


Figure 6.4: The graph representation of the I_c^f-preproof of $\{e'_{13}(u_1,u_2,u_3)\}$.

The normalisation process of the I_c^f-preproof applies the transformation from Figure 6.2 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 mpo 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 21 *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 6.2.4. 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 6.2.4. 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 6.2.4.

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.
```

6.4 Automatic certification of SPIKE proofs

The user can also interact with the SPIKE prover by the means of i) extra sections, for example `use: nats;`[17] 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`, such as `-coqc_spec`, `-coqc`, and `-dracula`.

In an automatic way, SPIKE can prove the conjecture (6.15) from Section 6.2 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.

6.5 Conclusions

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 *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

[17] To be added just after the `specification` section. More details about SPIKE are given in Chapter 4.

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 (6.15) 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.

Part III

Bridges with Other Reasoning Techniques

7

Cyclic Induction Reasoning for FOL with Inductive Definitions

Sommaire

7.1 The logical framework 90
7.1.1 Checking the soundness of pre-proofs 92
7.2 Defining the ordering-based checking criteria 93
7.2.1 Normalising pre-proof trees 93
7.2.2 Building the digraph of a pre-proof tree-set 95
7.2.3 Defining the ordering and derivability conditions 97
7.3 Implementation 101
7.4 Converting cyclic to Noetherian induction reasoning 102
7.4.1 The Cyclist proof 102
7.4.2 The conversion procedure 103
7.4.3 Experimental results 105
7.5 Conclusions 106

Motivation. CLKID$^\omega$ [Brotherston and Simpson, 2011] is the *de facto* standard sequent-based cyclic inference system for performing lazy induction reasoning on specifications based on first-order logic with inductive definitions (FOL_{ID}). The CLKID$^\omega$ proofs are represented as finite derivation trees with nodes labelled by sequents. A particular feature is that cycles can be built by establishing connections between terminal and non-terminal nodes labelled with identical sequents. The soundness of CLKID$^\omega$ proofs is entailed from some global trace condition by using Infinite Descent induction arguments [Wirth, 2004]. This condition requires that, for every infinite path in the cyclic derivation of a false sequent, all successive steps starting from some point are decreasing and certain steps occurring infinitely often are strictly decreasing w.r.t. some semantic ordering.

CLKID$^\omega$ has been implemented in the Cyclist prover [Brotherston *et al.*, 2012]. Since the global trace condition is an ω-regular property, Cyclist can check it during the proof construction or *post hoc* as an inclusion between two Büchi automata by calling an external model checker. It turns out that the inclusion test may be costly. Indeed, for any proof P, the approach requires the construction of the automaton complementary to that accepting strings over infinite progressing traces in P, based on a complementation method for Büchi automata as described in [Kupferman and Vardi, 2001]. The method ensures that, for every automaton with n states, the generated complementary automaton has at least $2^{O(n \log n)}$ states [Michel, 1988]. In case of failure of the inclusion test, previous proof steps should be reconsidered, requiring that existing connections be broken, proof steps cancelled or different inference rules applied. Hence, it may happen that the test be executed several times during the proof construction. For the proofs of the toy examples from [Brotherston *et al.*, 2012], the percentage of time taken by the soundness check include values from 0% to 44%.

Example 22 *Following[Brotherston* et al.*, 2012], the predicates P and Q from Example 1 can be specified in FOL_{ID} using the following productions:*

$$\Rightarrow P(0) \qquad (7.1)$$

$$P(x) \wedge Q(x, s(x)) \Rightarrow P(s(x)) \qquad (7.2)$$

$$\Rightarrow Q(x, 0) \qquad (7.3)$$

$$P(x) \wedge Q(x, y) \Rightarrow Q(x, s(y)) \qquad (7.4)$$

where 0 *and* s *are the usual constructor symbols for naturals. Similarly,* N *is defined as:*

$$\Rightarrow N(0) \qquad (7.5) \qquad\qquad N(x) \Rightarrow N(s(x)) \qquad (7.6)$$

According to Table 1 from [Brotherston et al., 2012], Cyclist *can prove the sequent* $N(x), N(y) \vdash Q(x, y)$ *in about half a second, by building a proof tree with only 13 nodes. The validation process required 181 calls to the external model checker, among which 171 calls are failing.* 31% *of the time is spent on the soundness check.*

Our approach. A different approach for checking the soundness of derivations built with a restricted version of CLKID$^\omega$, denoted by CLKID$^\omega_N$, is based on ordering constraints. It is inspired from the material presented in Chapter 6 about performing cyclic Noetherian induction to check inductive consequences of conditional specifications. The proofs generated by this approach are normalized to sets of tree derivations and represented as directed graphs (for short, digraphs) allowing some terminal nodes to be connected to root nodes. The minimal cycles resulting by following the arrows in the digraph are denoted as cyclic lists of paths leading a root to a terminal node in the same tree derivation. The ordering constraints for checking the proof soundness involve only comparisons between instances of root formulas. The main advantages of our approach are twofold: i) the worst-case time complexity of the validation procedure is polynomial, and ii) the proofs can be certified by Coq using the approach described in Chapter 6.

Structure of the chapter. Mostly based on [Stratulat, 2017a, Stratulat, 2018], the chapter is structured in five sections. Section 7.1 is a quick presentation to the logical framework based on FOL$_{\text{ID}}$ and an introduction to CLKID$^\omega_N$. In Section 7.2, we introduce the normalized form of CLKID$^\omega_N$ pre-proofs and define ordering and derivability constraints that guarantee the global trace condition. The implementation of the method is described in Section 7.3. A comparison with the automata-based method is given. On the other hand, we give an example that shows that our procedure is semi-decidable. In Section 7.4, we show that FOL$_{\text{ID}}$ pre-proofs can be dually represented as cyclic and Noetherian induction proofs. The conclusions are given in the last section.

7.1 The logical framework

The logical setting relies on FOL$_{\text{ID}}$ with equality, as presented, e.g., in [Brotherston, 2006, Brotherston and Simpson, 2011].

Syntax. We assume that Σ is a (countable) language built on a finite alphabet of arity-fixed function symbols $\mathcal{F}$ and predicate symbols, and $\mathcal{V}$ an enumerable set of variables. Each predicate symbol is either *inductive* (i.e., defined by axioms as below) or *ordinary* (i.e., not inductive). $P(t_1, \ldots, t_n)$ is an *inductive atom*, where P is an inductive predicate symbol and $t_1, \ldots, t_n$ are terms. $FV(S)$ denotes the set of free variables from the set of formulas S.

Each inductive predicate symbol P is defined by a finite inductive definition set of productions (axioms) consisting of implication formulas of the form

$$(\textstyle\bigwedge_{m=1}^{h} Q_m(\overline{u}_m) \wedge \bigwedge_{m=1}^{l} P_{i_m}(\overline{t}_m)) \Rightarrow P(\overline{t}), \qquad (7.7)$$

where $h, l, i_1, \ldots, i_l$ are naturals and $Q_1, \ldots, Q_h$ (resp., $P_{i_1}, \ldots, P_{i_l}$) are ordinary (resp., inductive) predicate symbols. (7.7) is an *unconditional* production if $h = 0$ and $l = 0$. If not, (7.7) is a *conditional* production and $\bigwedge_{m=1}^{h} Q_m(\overline{u}_m) \wedge \bigwedge_{m=1}^{l} P_{i_m}(\overline{t}_m)$ is its *condition*. Φ denotes the set of productions defining each inductive predicate symbol.

The CLKID$^\omega_N$ inference system. CLKID$^\omega_N$ is built from a finite set of inference rules that process *sequents*. A sequent is a logical construction of the form $\Gamma \vdash \Delta$, where Γ and Δ are finite multisets of first-order formulas and referred to as *antecedents* and *succedents*, respectively. An *inference rule* transforms a sequent, called *conclusion*, into a (potentially empty) multiset of sequents, called *premises*; they are separated by a horizontal line followed by the name of the rule. Most of the CLKID$^\omega_N$ inference rules transform one (principal) formula from the conclusion. In this case, it is explicitly represented in the sequent. A more detailed presentation of the sequent calculus can be found elsewhere, e.g., [Negri and v. Plato, 2001].

CLKID$^\omega_N$ consists of the rules contained by the LK system [Gentzen, 1935] and displayed in Figure 7.1, the rules that process equalities from Figure 7.2, as well as the 'unfold' and 'case' rules. $(= L)$ is an instance of the more general CLKID$^\omega$ version where x can also be a non-variable term.

$$\frac{}{\Gamma \vdash \Delta}\ \Gamma \cap \Delta \neq \emptyset\ (Ax) \qquad \frac{\Gamma' \vdash \Delta'}{\Gamma \vdash \Delta}\ \Gamma' \subseteq \Gamma, \Delta' \subseteq \Delta\ (Wk) \qquad \frac{\Gamma \vdash F, \Delta \quad \Gamma \vdash G, \Delta}{\Gamma \vdash F \wedge G, \Delta}\ (\wedge R)$$

$$\frac{\Gamma \vdash F, \Delta}{\Gamma, \neg F \vdash \Delta}\ (\neg L) \qquad \frac{\Gamma, F \vdash \Delta}{\Gamma \vdash \neg F, \Delta}\ (\neg R) \qquad \frac{\Gamma, F \vdash \Delta \quad \Gamma, G \vdash \Delta}{\Gamma, F \vee G \vdash \Delta}\ (\vee L)$$

$$\frac{\Gamma \vdash F, G, \Delta}{\Gamma \vdash F \vee G, \Delta}\ (\vee R) \qquad \frac{\Gamma, F, G \vdash \Delta}{\Gamma, F \wedge G \vdash \Delta}\ (\wedge L) \qquad \frac{\Gamma \vdash F, F, \Delta}{\Gamma \vdash F, \Delta}\ (contrR) \qquad \frac{\Gamma \vdash \Delta}{\Gamma[\theta] \vdash \Delta[\theta]}\ (Subst)$$

$$\frac{\Gamma \vdash F, \Delta \quad \Gamma, G \vdash \Delta}{\Gamma, F \Rightarrow G \vdash \Delta}\ (\Rightarrow L) \qquad \frac{\Gamma, F, F \vdash \Delta}{\Gamma, F \vdash \Delta}\ (contrL) \qquad \frac{\Gamma, F[\{\overline{x} \mapsto \overline{t}\}] \vdash \Delta}{\Gamma, \forall \overline{x} F \vdash \Delta}\ (\forall L)$$

$$\frac{\Gamma \vdash F, \Delta}{\Gamma \vdash \forall \overline{x} F, \Delta}\ \overline{x} \cap FV(\Gamma \cup \Delta) = \emptyset\ (\forall R) \qquad \frac{\Gamma, F \vdash \Delta}{\Gamma, \exists \overline{x} F \vdash \Delta}\ \overline{x} \cap FV(\Gamma \cup \Delta) = \emptyset\ (\exists L)$$

$$\frac{\Gamma \vdash F, \Delta \quad \Gamma, F \vdash \Delta}{\Gamma \vdash \Delta}\ (Cut) \qquad \frac{\Gamma \vdash F[\{\overline{x} \mapsto \overline{t}\}], \Delta}{\Gamma \vdash \exists \overline{x} F, \Delta}\ (\exists R) \qquad \frac{\Gamma, F \vdash G, \Delta}{\Gamma \vdash F \Rightarrow G, \Delta}\ (\Rightarrow R)$$

Figure 7.1: Sequent-based rules for classical first-order logic.

$$\frac{}{\Gamma \vdash t = t, \Delta}\ (= R) \qquad \frac{\Gamma[\{x \mapsto u\}] \vdash \Delta[\{x \mapsto u\}]}{\Gamma, x = u \vdash \Delta}\ x \text{ is not a variable of } u\ (= L)$$

Figure 7.2: Sequent-based rules for equality reasoning.

The unfold rule unrolls the definition of the inductive symbol to transform some succedent atom of a sequent. We denote the unfolding of $P(\overline{t}')$ with the production (7.7), when $P(\overline{t}') \equiv P(\overline{t})[\sigma]$, by

$$\frac{\Gamma \vdash Q_1(\overline{u}_1)[\sigma], \Delta \quad \ldots \quad \Gamma \vdash Q_h(\overline{u}_h)[\sigma], \Delta \qquad \Gamma \vdash P_{i_1}(\overline{t}_1)[\sigma], \Delta \quad \ldots \quad \Gamma \vdash P_{i_l}(\overline{t}_l)[\sigma], \Delta}{\Gamma \vdash P(\overline{t'}), \Delta}\ (R.(7.7))$$

The *case* rule is a left-introduction operation for inductive predicate symbols:

$$\frac{\text{case distinctions}}{\Gamma, P(s_1, \ldots, s_n) \vdash \Delta}\ (Case\ P)$$

Every production, of the form (7.7) such that $\overline{t} \equiv (t_1, \ldots, t_n)$, produces the *case distinction*

$$\Gamma, s_1 = t_1, \ldots, s_n = t_n, Q_1(\overline{u}_1), \ldots, Q_h(\overline{u}_h), P_{i_1}(\overline{t}_1), \ldots, P_{i_l}(\overline{t}_l) \vdash \Delta \tag{7.8}$$

Each variable y from (7.7) is fresh w.r.t. the *free variables* from the conclusion of the rule (y can be renamed to a fresh variable, otherwise). $P_{i_1}(\overline{t}_1), \ldots, P_{i_l}(\overline{t}_l)$ are *case descendants* of $P(s_1, \ldots, s_n)$.

CLKID$^\omega_N$ pre-proof trees. A *derivation tree* for some sequent S is built by successively applying inference rules starting from S. The terminal nodes in the tree can be either leaves or buds. A *leaf* is labelled by a sequent that is the conclusion of a 0-premise inference rule. A *bud* is every node labelled by a sequent that is the conclusion of no rule. For each bud, there is a *companion*, i.e., an internal node having the same sequent labelling. If a companion is annotated by some sign (e.g., † or $*$), then the buds related to it are uniquely annotated by that sign followed by a number.

Definition 5 (pre-proof tree, induction function for tree) *The pair ($\mathcal{D}$, $\mathcal{R}$) denotes a* pre-proof tree *of some sequent S, where $\mathcal{D}$ is a finite derivation tree whose root is labelled by S and $\mathcal{R}$ is a defined* induction function *assigning a companion to every bud in $\mathcal{D}$.*

Example 23 *A CLKID$^\omega_N$ pre-proof tree of $N(x), N(y) \vdash R(x, y)$ is*

$$\dfrac{\dfrac{}{Ny \vdash R(0,y)}\,(R.(7.9))}{\dfrac{\dfrac{}{\vdash R(0,0)}\,(R.(7.9)) \quad \dfrac{\dfrac{Nx' \vdash R(x',0)\ (\dagger\mathbf{1})}{Nx'' \vdash R(x'',0)}\,(Subst)}{Nx'' \vdash R(sx'',0)}\,(R.(7.10))}{\dfrac{\underline{Nx'} \vdash R(x',0)\ (\dagger)}{Nx' \vdash R(sx',0)}\,(R.(7.10))}\,(Case\ N) \quad \dfrac{\dfrac{\dfrac{Nx, Ny \vdash R(x,y)\ (*\mathbf{1})}{Nssx', Ny' \vdash R(ssx',y')}\,(Subst)}{Nx', Ny' \vdash R(ssx',y')}\,(Cut)}{Nx', Ny' \vdash R(sx',sy')}\,(R.(7.11))}{Nx', \underline{Ny} \vdash R(sx',y)}\,(Case\ N)}{\underline{Nx}, Ny \vdash R(x,y)\ (*)}\,(Case\ N)$$

where the inductive predicate R is defined, as in [Brotherston et al., 2012], by the productions

$$\Rightarrow R(0,y) \qquad (7.9) \qquad\qquad R(x,0) \Rightarrow R(sx,0) \qquad (7.10) \qquad\qquad R(ssx,y) \Rightarrow R(sx,sy) \qquad (7.11)$$

For lack of horizontal space, we have unambiguously omitted the parentheses and commas when denoting some natural and atom $N(t)$, i.e., $s(t)$ (resp., $N(t)$) becomes $s\underline{t}$ (resp., $N\underline{t}$), where $\underline{t}$ is the notation of t without parentheses. This alternative notation will be used in the following, when necessary.

The double line means that $(= L)$ was applied on each premise of $(Case)$. The (Cut) premise $Nx' \vdash Nssx'$ is suppressed on the right-hand branch as in Example 6 of [Brotherston et al., 2012]. The principal formula for each $(Case)$ application is underlined. Finally, the induction function $\mathcal{R}$ is defined such that the companion of the bud denoted by $(*1)$ (resp., $(\dagger 1)$) is $(*)$ (resp., $(\dagger)$).

Semantics. The semantics for FOL_{ID} with equality is defined as in [Brotherston and Simpson, 2011]. Prefixed points of a monotone operator issued from Φ [Aczel, 1977] help to interpret inductive predicates. A standard model for (Σ, Φ) is a first-order structure defined by the least prefixed point, approached by an iteratively built *approximant* sequence.

Definition 6 (validity of a sequent) *Let M be a standard model for (Σ, Φ), $\Gamma \vdash \Delta$ a sequent and ρ a valuation which interprets in M the free variables from the sequent. We write $\Gamma \models^M_\rho \Delta$ if whenever G holds in M using ρ, for all $G \in \Gamma$, there is some $D \in \Delta$ that holds in M using ρ. We say that $\Gamma \vdash \Delta$ is M-*true *if $\Gamma \models^M_\rho \Delta$, for every ρ. When M is implicit from the context,* true *is used instead of M-true.*

A rule is *sound*, or preserves the validity, if its conclusion is true whenever its premises are true. Hence, the conclusion of every 0-premise sound rule is true.

Theorem 18 *The CLKID^ω_N inference rules are sound.*

Proof CLKID^ω_N is an instance of CLKID^ω which has been shown sound in [Brotherston and Simpson, 2011]. □

Definition 7 (sound pre-proof tree) *A pre-proof tree of a sequent S is* sound *if S is true.*

7.1.1 Checking the soundness of pre-proofs

Not every pre-proof tree is sound. A very simple example of unsound pre-proof tree can be built for every false sequent S by firstly adding a copy of some antecedent formula using $(contrL)$ then deleting it using (Wk). Since the resulting sequent is identical to S, its node is a bud. This finishes the pre-proof tree.

Before presenting the global trace condition, which is a sufficient condition to ensure the soundness of CLKID^ω_N proofs, we introduce some preliminary definitions. We denote by $S(N)$ the sequent labelling any node N. A *path* is a list $[N^0, N^1, \ldots]$ of nodes in a pre-proof tree such that, for all $i \geq 0$, $S(N^{i+1})$ is either one of the premises of the rule applied on $S(N^i)$ if N^i is not a terminal node, or $S(\mathcal{R}(N^i))$ if N^i is a bud.

Definition 8 (Trace, Progress point [Stratulat, 2017a]) *Let $(\mathcal{D}, \mathcal{R})$ be a CLKID^ω_N pre-proof tree and let $[N^0, N^1, \ldots]$ be one of its infinite paths and denoted by l. A* trace *following l is a sequence $(\tau_i)_{i \geq 0}$ of inductive antecedent atoms (IAAs) such that, for all i, we have that N^i is labelled by $\Gamma_i \vdash \Delta_i$ and:*

1. *τ_i is some $P_{j_i}(\overline{t_i}) \in \Gamma_i$;*

2. *if* $\Gamma_i \vdash \Delta_i$ *is the conclusion of* $(Subst)$ *then* $\tau_i = \tau_{i+1}[\theta]$*, where* θ *is the substitution used by the LK's* $(Subst)$ *rule defined as:*

$$\frac{\Gamma \vdash \Delta}{\Gamma[\theta] \vdash \Delta[\theta]}\ (Subst)$$

3. *if* $\Gamma_i \vdash \Delta_i$ *is the conclusion of* $(= L)$ *having* $t = u$ *as principal formula, there is a formula* F *such that* $\tau_i = F$ *and* $\tau_{i+1} = F[\{t \mapsto u\}]$*;*

4. *if* $\Gamma_i \vdash \Delta_i$ *is the conclusion of a* $(Case)$ *rule then either a)* $\tau_{i+1} = \tau_i$*, if* τ_i *is not the principal formula of the rule instance, or b)* τ_i *is the principal formula and* τ_{i+1} *is a case descendant of* τ_i*. In the latter case,* i *is said to be a* progress point *of the trace;*

5. *if* $\Gamma_i \vdash \Delta_i$ *is the conclusion of any other rule then* $\tau_{i+1} = \tau_i$*.*

Remark 1 *Non-equality relations between (instances of)* τ_i *and* τ_{i+1} *in the above definition are possible only when* i *is a progress point.*

Remark 2 *Condition 3 is an abbreviated form of the case dealing with* $(= L)$ *in Definition 5.4 from [Brotherston and Simpson, 2011], by applying the discussed restrictions to* $(= L)$*, i.e., if* $\Gamma_i \vdash \Delta_i$ *is the conclusion of* $(= L)$*, of the form* $\Gamma[\{x \mapsto t; y \mapsto u\}], t = u \vdash \Delta[\{x \mapsto t; y \mapsto u\}]$ *and having* $t = u$ *as principal formula, there is a formula* F' *such that* $\tau_i = F'[\{x \mapsto t; y \mapsto u\}]$ *and* $\tau_{i+1} = F'[\{x \mapsto u; y \mapsto t\}]$ *under the following conditions:* $y \notin FV(\Gamma_i \backslash \{t = u\} \vdash \Delta_i)$*,* t *is a free variable not occurring in* u *and* $t \notin FV(\Gamma \vdash \Delta)$*.*

An *infinitely progressing* trace is a trace with infinitely many progress points. In [Brotherston and Simpson, 2011], it has been shown that a pre-proof tree $\mathcal{D}$ is sound if it satisfies the *global trace condition*, i.e., for every infinite path p in $\mathcal{D}$, there is an infinitely progressing trace following some tail of p.

In the next section, we introduce an approach similar to that used for building Noetherian (well-founded) induction-based proofs [Stratulat, 2012] to check the global trace condition.

7.2 Defining the ordering-based checking criteria

We define ordering and derivability conditions to be satisfied by the digraph representing some normal form of the pre-proof. The normalisation procedure transforms the pre-proof into a set of pre-proof trees, for short *pre-proof tree-sets*, such that the root of the pre-proof is among the roots of the trees from the normal form. If the sequent labelling the root of the pre-proof is false, one can build an infinite path in the digraph, whose nodes are labelled by false sequents and for which there is an infinite progressing trace following some tail of it.

7.2.1 Normalising pre-proof trees

The normalisation process consists in the exhaustive application of the following three operations.

The first operation applies on an internal node labelled by some premise of $(Subst)$, of the form

$$\vdots$$
$$\frac{\dfrac{}{\Gamma \vdash \Delta}}{\Gamma[\sigma] \vdash \Delta[\sigma]}\ (Subst)$$
$$\vdots$$

The result is displayed in Figure 7.3. The internal node is duplicated and the subtree derivation rooted by it is detached to become a new tree derivation. At the end, we get two distinct pre-proof trees. The two occurrences of the duplicated node establish a new bud-companion relation.

$$\dfrac{\Gamma \vdash \Delta\ (*1)}{\Gamma[\sigma] \vdash \Delta[\sigma]}\,(Subst) \qquad\qquad \dfrac{\vdots}{\Gamma \vdash \Delta\ (*)}$$

$$\vdots \qquad\qquad \text{(new tree)}$$

Figure 7.3: The result of the first operation.

The second operation applies on a non-root companion which is duplicated and the subtree derivation rooted by it becomes a new pre-proof tree. The result is displayed in Figure 7.4. The sequent labelling the copy of the companion $(*)$ becomes the conclusion of a new $(Subst)$ rule. The substitution used by the new $(Subst)$ rule is chosen such that its premise labels a new bud node labelled by the same sequent as the conclusion, e.g., the *empty* substitution. The new bud node will have (*) assigned as companion.

$$\dfrac{\Gamma \vdash \Delta\ (*1)}{\Gamma \vdash \Delta}\,(Subst) \qquad\qquad \dfrac{\vdots}{\Gamma \vdash \Delta\ (*)}$$

$$\vdots \qquad\qquad \text{(new tree)}$$

Figure 7.4: The result of the second operation.

The last operation applies on a bud node labelled by some sequent that is the premise of a rule r different from $(Subst)$ such that

$$\dfrac{\Gamma \vdash \Delta\ (*1)}{\Gamma' \vdash \Delta'}\,r \qquad \text{is transformed to} \qquad \dfrac{\dfrac{\Gamma \vdash \Delta\ (*1)}{\Gamma \vdash \Delta}\,(Subst)}{\Gamma' \vdash \Delta'}\,r$$

$$\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots$$

Let $(*)$ denote the companion of the bud node. A new application of $(Subst)$ with the empty substitution was performed on the bud sequent such that the node labelled by its premise becomes the new bud node whose companion is $(*)$.

Compared with a similar 2-operation normalisation procedure presented in [Stratulat, 2017a], only the first operation is shared by the two procedures. The other operation of the procedure from [Stratulat, 2017a] applies on non-root companions but does not include the $(Subst)$-step from Figure 7.4. There is no equivalent for the third operation.

The following properties, related to the normalisation process and the resulting normal form as given by Lemmas 7 and 8, are satisfied.

Lemma 7 (termination) *The normalisation process terminates.*

Proof The number of nodes that can be processed by the three operations is finite, for every pre-proof tree. In addition, it decrements after applying each operation. □

The induction function is extended to allow new bud-companion relations between nodes from different pre-proof trees.

Definition 9 (rb-path, IH-node) *An* rb-path *is a path of the form* $[R, \ldots, H, B]$ *that leads the root* R *to a bud* B *in some pre-proof tree of a pre-proof tree-set such that* B *is the only bud in the path. We will call* H *an inductive hypothesis node (for short,* IH-node*).*

A path in a pre-proof tree-set $(\mathcal{MD}, \mathcal{MR})$ is a list $[N^0, N^1, \ldots]$ of nodes in $\mathcal{MD}$ such that, for all $i \geq 0$, $S(N^{i+1})$ is one of the premises of the rule applied on $S(N^i)$ if N^i is an internal node, or $S(\mathcal{MR}(N^i))$ if N^i is a bud.

Lemma 8 *The normalisation of any pre-proof $(\mathcal{D}, \mathcal{R})$ of a sequent S builds a pre-proof tree-set ($\mathcal{MD}$, $\mathcal{MR}$) for which*

1. *all companions are root nodes,*
2. *there is a pre-proof tree rooted by a node labelled by S, and*
3. *each of its rb-paths $[R, \ldots, B]$ has B as the only node that is labelled by the premise of a $(Subst)$ rule. A node is a $(Subst)$-node if and only if it is an (IH)-node.*

Proof Claim 1) follows from the exhaustive application of the second operation.

Claim 2) holds because the first operation duplicates only non-root nodes and the third operation expands bud nodes, so the root nodes do not change. If S labels the root node of a pre-proof tree t having a non-root companion n, t will be processed by the second operation applied on n but will still have its root labelled by S.

Claim 3) holds by the construction of the normal forms. □

Example 24 *The second operation can be applied on the non-root companion from Example 23, denoted by $(*)$, to give the following normalised pre-proof tree-set:*

$$\dfrac{\dfrac{}{Ny \vdash R(0,y)}\,(R.(7.9)) \qquad \dfrac{\dfrac{\dfrac{\underline{Nx'} \vdash R(x',0)\ (\dagger\mathbf{1})}{\underline{Nx'} \vdash R(x',0)}\,(Subst)}{Nx' \vdash R(sx',0)}\,(R.(7.10)) \qquad \dfrac{\dfrac{Nx, Ny \vdash R(x,y)\ (*)}{Nssx', Ny' \vdash R(ssx', y')}\,(Subst)}{Nx', Ny' \vdash R(ssx', y')}\,(Cut)}{Nx', Ny' \vdash R(sx', sy')}\,(R.(7.11))}{Nx', \underline{Ny} \vdash R(sx', y)}\,(Case\ N)}{\underline{Nx}, Ny \vdash R(x,y)\ (*)}\,(Case\ N)$$

$$\dfrac{\dfrac{}{\vdash R(0,0)}\,(R.(7.9)) \qquad \dfrac{\dfrac{Nx' \vdash R(x',0)\ (\dagger)}{Nx'' \vdash R(x'',0)}\,(Subst)}{Nx'' \vdash R(sx'',0)}\,(R.(7.10))}{\underline{Nx'} \vdash R(x',0)\ (\dagger)}\,(Case\ N)$$

7.2.2 Building the digraph of a pre-proof tree-set

Any pre-proof tree-set can also be represented as a *digraph* of sequents built from the nodes of its tree-set. The digraph associated to a pre-proof tree-set ($\mathcal{MD}$, $\mathcal{MR}$) is crucial in our setting to check whether ($\mathcal{MD}$, $\mathcal{MR}$) is a proof tree-set. Its edges are arrows built as follows:

- a *forward* arrow leads a node N^1 to a node N^2 if there is a rule that was applied on the sequent labelling N^1 and the sequent labelling N^2 is a premise of the rule;
- a *back-link* (or backward arrow) starts from a bud and ends to its companion.

Some arrows will be annotated by substitutions. Each forward arrow, starting from a $(= L)$-node whose principal formula is $x = u$, is annotated by the *equality substitution* $\{x \mapsto u\}$. The forward arrow starting from a node N that is different from $(= L)$- and $(Subst)$-nodes is annotated with the *identity substitution* for $S(N)$, which maps the free variables from $S(N)$ to themselves. Finally, the forward arrows starting from $(Subst)$-nodes and the back-links are not annotated. They help to build infinite paths but do not play any role when defining the soundness constraints.

By abuse of notation, a *path* in a digraph is a (potentially infinite) list of nodes built by following the arrows in the digraph. An *rb-path* is any path leading a root to some bud node and does not have other bud nodes. **Unless otherwise stated, we will consider only rb-paths in the digraphs associated to *normalised* pre-proof tree-sets.**

Remark 3 *According to Lemma 8, the bud node B of any such rb-path is the only node in the rb-path for which $S(B)$ is the premise of a $(Subst)$ rule.*

Definition 10 (cumulative substitution) *An rb-path* $[N^1, \ldots, N^n, B]$ *(*$n > 0$*) can be annotated by the* cumulative substitution $\sigma_{id}^{all}\sigma_1 \cdots \sigma_{n-1}$*, where* σ_i *is the substitution annotating the forward arrow leading* N_i *to* N_{i+1}*, for each* $i \in [1..n-1]$*, and* σ_{id}^{all} *is the* overall *identity substitution* $\cup_{N \in [N^1,\ldots,N^{n-1}]}\{x \mapsto x \mid x \in FV(S(N))\}$.

A list of sequents $[S_1, \ldots, S_n]$ $(n > 0)$ is *admissible* if either i) it is a singleton $(n = 1)$, or ii) for every $i \in [2..n]$, S_i is the premise of some rule whose conclusion is S_{i-1}. By construction, the list of sequents labelling the nodes from every path from the digraph associated to a pre-proof tree-set is admissible.

Lemma 9 *Let* $[N^1, \ldots, N^{n-1}, N^n, B]$ *be an rb-path. We define its* cumulative *list* l_c *as* $[S(N^1)[\theta^c_{(1,n)}], \ldots, S(N^{n-1})[\theta^c_{(n-1,n)}], S(N^n), S(B)]$*, where* $\theta^c_{(i,n)}$ *is the cumulative substitution for* $[N^i, \ldots, N^{n-1}, N^n]$*. Then, the following properties hold:*

1. l_c *is admissible, and*
2. *the rule applied on each* $S(N^i)$ *is also applicable on* $S(N^i)[\theta^c_{(i,n)}]$*,* $\forall i \in [1..n-1]$*, if it is different from* $(= L)$*. If the rule is* $(= L)$*, the* $(= L)$*-step can be replaced by a* (Wk)*-step, where the LK's* (Wk) *rule is defined as*

$$\frac{\Gamma' \vdash \Delta'}{\Gamma \vdash \Delta}\,(Wk)\ \textit{if}\ \Gamma' \subseteq \Gamma, \Delta' \subseteq \Delta$$

Proof We will perform induction on n. If $n = 1$, then $N^n \equiv N^1$ and $[S(N^1)]$ is a singleton, hence it is admissible.

If $n > 1$, let p denote the path $[N^1, \ldots, N^{n-1}, N^n, B]$. By induction hypothesis, we assume that $[S(N^1)[\theta^c_{(1,n-1)}], \ldots, S(N^{n-2})[\theta^c_{(n-2,n-1)}], S(N^{n-1})]$ is admissible, where $\theta^c_{(i,n-1)}$ $(i \in [1..n-2])$ is the cumulative substitution annotating $[N^i, \ldots, N^{n-1}]$ and the rules applied on $S(N^i)[\theta^c_{(i,n-1)}]$ and $S(N^i)$ are the same. We denote by $\theta^c_{(n-1,n-1)}$ the identity substitution for $S(N^{n-1})$.

Let $\theta^c_{(i,n)}$ be the cumulative substitution annotating $[N^i, \ldots, N^{n-1}, N^n]$, for all $i \in [1..n-1]$. Let also θ be the substitution annotating the forward arrow leading N^{n-1} to N^n, which can be either an identity substitution, or an equality substitution. In the first case, for every $i \in [1..n-1]$, $\theta^c_{(i,n)}$ is i) $\theta^c_{(i,n-1)} \cup \{x \mapsto x \mid x \in \overline{x}\}$ if the rule applied on $S(N^{n-1})$ is the LK's rule $(\forall R)$ or $(\exists L)$, defined below:

$$\frac{\Gamma \vdash F, \Delta}{\Gamma \vdash \forall \overline{x} F, \Delta}\,(\forall R)\ \text{if}\ \overline{x} \cap FV(\Gamma \cup \Delta) = \emptyset$$

$$\frac{\Gamma, F \vdash \Delta}{\Gamma, \exists \overline{x} F \vdash \Delta}\,(\exists L)\ \text{if}\ \overline{x} \cap FV(\Gamma \cup \Delta) = \emptyset$$

and $\overline{x}$ is the vector of new free variables introduced by these rules, or ii) $\theta^c_{(i,n-1)}$, otherwise. Since $S(N^i)[\theta^c_{(i,n-1)}] \equiv S(N^i)[\theta^c_{(i,n)}]$ by induction hypothesis, we can apply the same rules on $S(N^i)[\theta^c_{(i,n)}]$ and $S(N^i)$, hence the list $[S(N^1)[\theta^c_{(1,n)}], \ldots, S(N^{n-1})[\theta^c_{(n-1,n)}], S(N^n)]$ is admissible. $[S(N^1)[\theta^c_{(1,n)}], \ldots, S(N^{n-1})[\theta^c_{(n-1,n)}], S(N^n), S(B)]$ is also admissible since $S(B)$ is the premise of a $(Subst)$ rule whose conclusion is $S(N^n)$, by property 2) from Lemma 8.

For the second case, θ is an equality substitution. We have that $\theta^c_{(i,n)}$ equals $\theta^c_{(i,n-1)}\theta$, for all $i \in [1..n-1]$. Since the rule applied on a sequent can also be applied on every instance of it, we have that $[S(N^1)[\theta^c_{(1,n)}], \ldots, S(N^{n-1})[\theta^c_{(n-1,n)}], S(N^n)]$ is admissible; the rule applied on $S(N^i)$ can also be applied on $S(N^i)[\theta^c_{(i,n)}]$, for all $i \in [1..n-1]$. Notice that the $(= L)$ rule has $S(N^n)$ as premise when applied on $S(N^{n-1})[\theta^c_{(n-1,n)}\theta]$. Let us assume that $x = u$ is the principal formula of $S(N^{n-1})[\theta^c_{(n-1,n)}]$. Then, θ is $\{x \mapsto u\}$. On the one hand, $(= L)$ cannot be applied on $S(N^{n-1})[\theta^c_{(n-1,n)}\theta]$, whose principal formula is $u = u$, when u is a non-variable term. On the other hand, the generalised form of $(= L)$ from CLKID$^\omega$, defined in [Brotherston and Simpson, 2011] as

$$\frac{\Gamma[\{x \mapsto u; y \mapsto t\}] \vdash \Delta[\{x \mapsto u; y \mapsto t\}]}{\Gamma[\{x \mapsto t; y \mapsto u\}], x = u \vdash \Delta[\{x \mapsto t; y \mapsto u\}]}\ (= L) \quad ,$$

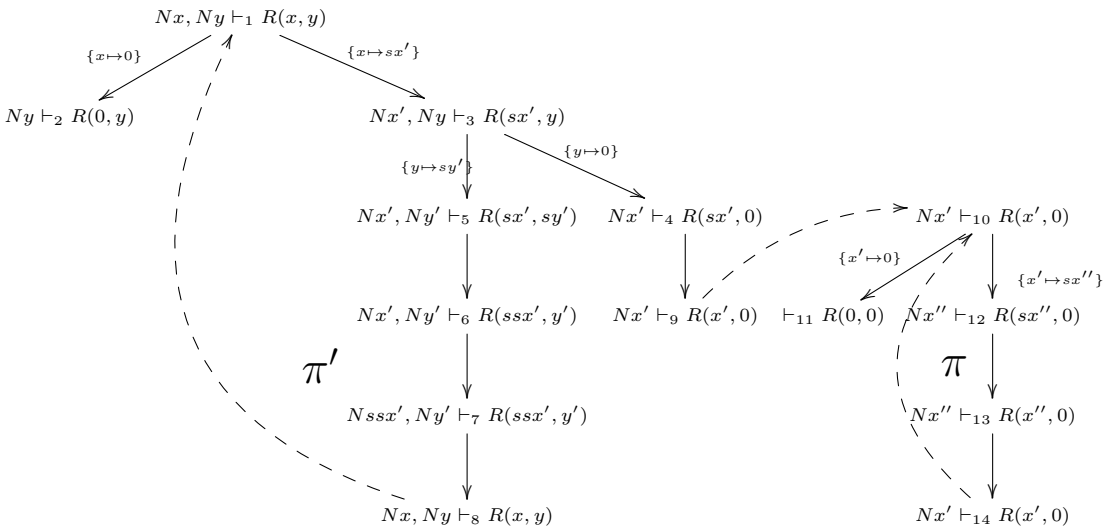
would replace u by u and delete $u = u$. If $S(N^{n-1})[\theta^c_{(n-1,n)}\theta]$ is of the form $\Gamma, u = u \vdash \Delta$, the same result can be achieved with CLKID$^\omega_N$ by applying (Wk) instead:

$$\frac{\Gamma \vdash \Delta}{\Gamma, u = u \vdash \Delta}\ (Wk)$$

So, the list $[S(N^1)[\theta^c_{(1,n)}], \ldots, S(N^{n-1})[\theta^c_{(n-1,n)}], S(N^n)]$ is admissible.
$[S(N^1)[\theta^c_{(1,n)}], \ldots, S(N^{n-1})[\theta^c_{(n-1,n)}], S(N^n), S(B)]$ is also admissible, as shown for the first case. □

A path has *cycles* if some nodes are repeated in the path. The set of *strongly connected components* (SCCs) of a digraph $\mathcal{P}$ of some pre-proof tree-set $(\mathcal{MD}, \mathcal{MR})$, for which every non-singleton SCC has at least one cycle, is a partition of $\mathcal{P}$. Additionally, if $\mathcal{P}$ is acyclic, each of its nodes is a singleton SCC.

Example 25 *The digraph of the normalised pre-proof tree-set from Example 24 is:*



The sequent labelling a node is annotated by the number of the node in the digraph. The digraph has two non-singleton SCCs: i) π:$\{N^{10}, N^{12}, N^{13}, N^{14}\}$, and ii) $\pi' : \{N^1, N^3, N^5, N^6, N^7, N^8\}$.

7.2.3 Defining the ordering and derivability conditions

The premises for defining the new soundness criterion are similar to [Stratulat, 2017a]. Let π be a SCC from $\mathcal{P}$ and $<_a$ an ordering stable under substitutions, defined over the set $\mathcal{S}$ of instances of the IAAs from the sequents labelling nodes inside π. Given a path p in π, we say that an IAA τ_j *derives from* an IAA τ_i using the trace $(\tau_k)_{(k \geq 0)}$ along p if $i < j$. Also, given two arbitrary substitutions γ and δ, we say that $\tau_j[\gamma]$ derives from $\tau_i[\delta]$ using $(\tau_k)_{(k \geq 0)}$ along p. $<_\pi$ is the multiset extension of $<_a$.

The ordering constraints from a multiset extension relation comparing two sequent instances can be combined with derivability constraints on IAAs to give the *$<_\pi$-derivability* relation, referred to as *ordering-derivability* when the ordering is not known. For this, we assume that every sequent S has associated a measure value (weight), denoted by A_S and represented by a multiset of IAAs of S.

Definition 11 ($<_\pi$-derivability) *Let N^i and N^j be two nodes occurring in some path p from π, and θ, δ be two substitutions. We define $A'_{S(N^i)[\theta]}$ (resp., $A'_{S(N^j)[\delta]}$) as the multiset, resulting from $A_{S(N^i)[\theta]}$ (resp., $A_{S(N^j)[\delta]}$) after the pairwise deletion of all common IAAs from $A_{S(N^i)[\theta]}$ and $A_{S(N^j)[\delta]}$. In addition, we*

assume that for each $l \in A_{S(N^j)[\delta]} \backslash A'_{S(N^j)[\delta]}$, *there is* $l' \in A_{S(N^i)[\theta]} \backslash A'_{S(N^i)[\theta]}$ *satisfying i)* $l \equiv l'$, *and ii)* l *is the unique literal from* $A_{S(N^j)[\delta]}$ *that derives from* l' *using some trace following* p.

Then, $S(N^j)[\delta]$ *is* $<_\pi$-derivable *from* $S(N^i)[\theta]$ *along* p *if for each* $l \in A'_{S(N^j)[\delta]}$ *there exists* $l' \in A'_{S(N^i)[\theta]}$ *such that* $l' >_a l$ *and* l *derives from* l' *using some trace following* p.

By the definition of $<_\pi$ as a multiset extension of $<_a$, the following results can be proved when considering some path in π.

Lemma 10 *If* S *is* $<_\pi$*-derivable from* S' *then* $A_S <_\pi A_{S'}$.

Proof By the definition of the ordering constraint in the $<_\pi$-derivability relation. □

Lemma 11 *The '*$<_\pi$*-derivability' relation is stable under substitutions and transitive.*

Proof Let S and S' be two sequents such that S is $<_\pi$-derivable from S' along some path p in π. By Lemma 10, $A_{S'} >_\pi A_S$. Since $<_\pi$ is stable under substitutions, we have that $A_{S'[\sigma]} >_\pi A_{S[\sigma]}$, for every substitution σ. According to Definition 11, the derivability relations between their IAAs do not change by instantiation operations. Therefore, $S[\sigma]$ is $<_\pi$-derivable from $S'[\sigma]$ along p. We conclude that the '$<_\pi$-derivability' relation is stable under substitutions.

To prove the transitivity property, let us assume three sequents S_1, S_2 and S_3 labelling nodes in a path p built by the concatenation of two paths p_1 and p_2 such that S_3 is $<_\pi$-derivable from S_2 along p_2 and S_2 is $<_\pi$-derivable from S_1 along p_1. We will try to prove that S_3 is $<_\pi$-derivable from S_1 along p.

Since S_3 is $<_\pi$-derivable from S_2 along p_2, by Definition 11 we have that

(i1) for each $l_3 \in A'_{S_3}$ there exists $l_2 \in A'_{S_2}$ such that $l_2 >_a l_3$ and l_3 derives from l_2 using some trace following p_2, and

(ii1) for each $l_3 \in A_{S_3} \backslash A'_{S_3}$, there is some $l_2 \in A_{S_2} \backslash A'_{S_2}$ such that $l_3 \equiv l_2$ and l_3 is the unique IAA that derives from l_2 using some trace following p_2,

where A'_{S_3} (resp., A'_{S_2}) is the multiset resulting from A_{S_3} (resp., A_{S_2}) after the pairwise deletion of all common IAAs from A_{S_3} and A_{S_2}. Also, since S_2 is $<_\pi$-derivable from S_1 along p_1, we have that

(i2) for each $l_2 \in A''_{S_2}$, there exists $l_1 \in A'_{S_1}$ such that $l_1 >_a l_2$ and l_2 derives from l_1 using some trace following p_1, and

(ii2) for each $l_2 \in A_{S_2} \backslash A''_{S_2}$, there is some $l_1 \in A_{S_1} \backslash A'_{S_1}$ such that $l_2 \equiv l_1$ and l_2 is the unique IAA that derives from l_1 using some trace following p_1,

where A''_{S_2} (resp., A'_{S_1}) is the multiset resulting from A_{S_2} (resp., A_{S_1}) after the pairwise deletion of all common IAAs from A_{S_2} and A_{S_1}. We have to check that for each $l_3 \in A''_{S_3}$, there exists $l_1 \in A''_{S_1}$ such that $l_1 >_a l_3$ and l_3 derives from l_1 using some trace following p, where A''_{S_3} (resp., A''_{S_1}) is the multiset resulting from A_{S_3} (resp., A_{S_1}) after the pairwise deletion of all common IAAs from A_{S_3} and A_{S_1}. Moreover, for each $l_3 \in A_{S_3} \backslash A''_{S_3}$, there is some $l_1 \in A_{S_1} \backslash A''_{S_1}$ such that $l_3 \equiv l_1$ and l_3 is the unique IAA that derives from l_1 using some trace following p. We consider the following cases:

1. If $l_3 \in A'_{S_3}$ there exists $l_2 \in A'_{S_2}$ such that $l_2 >_a l_3$ and l_3 derives from l_2 using some trace t_2 following p_2.
   (a) If $l_2 \in A''_{S_2}$ there exists $l_1 \in A'_{S_1}$ such that $l_1 >_a l_2$ and l_2 derives from l_1 by using some trace t_1 following p_1. Then $l_1 >_a l_3$ by the transitivity of $<_a$, so $l_1 \in A''_{S_1}$, $l_3 \in A''_{S_3}$ and l_3 derives from l_1 using the concatenation of t_1 and t_2 following p.
   (b) If $l_2 \in A_{S_2} \backslash A''_{S_2}$, there is $l_1 \in A_{S_1} \backslash A'_{S_1}$ such that $l_2 \equiv l_1$ and l_2 is the unique IAA that derives from l_1 by using some trace t_1 following p_1. Since $l_1 (\equiv l_2) >_a l_3$, we have that $l_1 \in A''_{S_1}$, $l_3 \in A''_{S_3}$ and l_3 derives from l_1 using the concatenation of t_1 and t_2 following p.

2. If $l_3 \in A_{S_3} \backslash A'_{S_3}$ there exists $l_2 \in A'_{S_2}$ such that $l_3 \equiv l_2$ and l_3 is the unique IAA that derives from l_2 using some trace t_2 following p_2.

   (a) If $l_2 \in A''_{S_2}$ there exists $l_1 \in A'_{S_1}$ such that $l_1 >_a l_2$ and l_2 derives from l_1 by using some trace t_1 following p_1. Then, $l_1 >_a (l_2 \equiv) l_3$, so $l_1 \in A''_{S_1}$, $l_3 \in A''_{S_3}$ and l_3 derives from l_1 using the concatenation of t_1 and t_2 following p.

   (b) If $l_2 \in A_{S_2} \backslash A''_{S_2}$ there exists $l_1 \in A'_{S_1}$ such that $l_1 \equiv l_2$ and l_2 is the unique IAA that derives from l_1 by using some trace t_1 following p_1. This means that $l_3 \in A_{S_3} \backslash A''_{S_3}$, $l_1 \in A_{S_1} \backslash A''_{S_1}$ with $l_1 \equiv (l_2 \equiv) l_3$ and l_3 derives from l_1 using the concatenation of t_1 and t_2 following p. In addition, l_3 is the unique IAA in A_{S_3} that derives from l_1.

□

The soundness criterion consists in checking if the sequents labelling (IH)-nodes from every non-singleton SCC, referred to as *induction hypotheses*, satisfy some constraints.

Definition 12 (induction hypothesis (IH), IH discharged by a SCC) *Let π be a non-singleton SCC and $[R, \ldots, H, B]$ an rb-path p in π. We say that the* induction hypothesis *(IH) $S(H)$ is* discharged *by π if $S(H)$ is $<_\pi$-derivable from $S(R)[\theta^c]$ along p, where θ^c is the cumulative substitution annotating p.*

Theorem 19 (soundness) *The sequents, labelling the roots from every normalised pre-proof tree-set whose non-singleton SCCs discharge their IHs, are true.*

Proof Let M be a standard model for (Σ, Φ) and assume a normalised pre-proof tree-set. Let also $\mathcal{P}$ denote its digraph whose non-singleton SCCs discharge their IHs. By contradiction, we assume that there exists a root node N such that $S(N)$ is false. We define a partial (well-founded) ordering $<_\mathcal{R}$ over the (finite number of) root nodes from $\mathcal{P}$ such that, for every two distinct root nodes N^1 and N^2, we have $N^1 <_\mathcal{R} N^2$ if i) N^1 and N^2 are not in the same SCC, and ii) N^1 can be joined from N^2 in $\mathcal{P}$.

By induction on $<_\mathcal{R}$, we consider the base case when N is a $<_\mathcal{R}$-minimal node. (The step case, when N is not a $<_\mathcal{R}$-minimal node, will not be detailed since it can be treated similarly by assuming that all $<_\mathcal{R}$-smaller root nodes are labelled by true sequents.) If N is included in a one-node SCC, N is also a leaf node. The only 0-premise rules are the LK's (Ax) rule as well as $(R.)$ when unfolding with unconditional axioms. In both cases, $S(N)$ is true which leads to a contradiction.

Let us now assume that N is a $<_\mathcal{R}$-minimal node from some non-singleton SCC π. We will analyse all possible scenarios and show that each of them leads to a contradiction. The tree t from $\mathcal{P}$ and rooted by N should have buds labelled by false sequents, otherwise $S(N)$ would be true. Let B be such a bud such that N^h is its companion and $[N, \ldots, H, B]$ is an rb-path in π. N^h should be a root node from π because N is $<_\mathcal{R}$-minimal; it is labelled by the false sequent $S(B)$. Since the CLKID^ω_N rules are sound, by Lemma 9, we conclude that the *cumulative instance* $S(N)[\theta_c]$ is false, where θ_c is the cumulative substitution for $[N, \ldots, H, B]$. π discharges its IHs, so we have that $S(B)[\delta_h] (\equiv S(H))$ is $<_\pi$-derivable from $S(N)[\theta_c]$, where δ_h is the substitution used by the $(Subst)$-step whose conclusion is $S(H)$. By Lemma 10, we have that $A_{S(N^h)[\delta_h]} <_\pi A_{S(N)[\theta_c]}$.

We perform a similar reasoning on N^h as for N. There is an rb-path $[N^h, \ldots, H', N^{f'}]$ such that the companion of $N^{f'}$ (in π) is $N^{h'}$ and $S(N^h)[\delta_h]$ shares false instances with $S(N^h)[\theta^c_1]$, where θ^c_1 is the cumulative substitution annotating $[N^h, \ldots, H', N^{f'}]$. By contradiction, we assume that no false instance of $S(N^h)[\delta_h]$ is shared. Then, one can build a finite bud-free pre-proof tree of $S(N^h)[\delta_h]$, by using only sound rules. Hence, $S(N^h)[\delta_h]$ is true, so contradiction. Therefore, there are two substitutions ϵ and τ such that $S(N^h)[\delta_h \epsilon] \equiv S(N^h)[\theta^c_1 \tau]$ and $S(N^h)[\theta^c_1 \tau]$ is false. Let $S(N^{h'})[\delta'_h] (\equiv S(H'))$ be the instance of $S(N^{h'})$ used as IH. Since it is discharged by π, we have that $A_{S(N^h)[\theta^c_1]} >_\pi A_{S(N^{h'})[\delta'_h]}$. From $A_{S(N)[\theta^c]} >_\pi A_{S(N^h)[\delta_h]}$ and the previous ordering constraint, we get $A_{S(N)[\theta^c \epsilon]} >_\pi A_{S(N^h)[\delta_h \epsilon]}$ and $A_{S(N^h)[\theta^c_1 \tau]} >_\pi A_{S(N^{h'})[\delta'_h \tau]}$, by the 'stability under substitutions' property of $<_\pi$. Hence,

$$A_{S(N)[\theta^c \epsilon]} >_\pi A_{(S(N^h)[\delta_h \epsilon]} \equiv) A_{S(N^h)[\theta^c_1 \tau]} >_\pi A_{S(N^{h'})[\delta'_h \tau]}$$

For similar reasons as given for $S(N^h)[\delta_h]$, we can show that $S(N^{h'})[\delta'_h\tau]$ is false, hence it can be treated similarly as $S(N^h)[\delta_h]$. And so on, the process can be repeated to produce an infinite strictly $<_\pi$-decreasing sequence s of measure values associated to instances of sequents labelling root nodes from π, of the form

$$A_{S(N)[\theta^c\epsilon\cdots]} >_\pi A_{S(N^h)[\theta_1^c\tau\cdots]} >_\pi A_{S(N^{h'})[\cdots]} >_\pi \cdots$$

We can associate to s the infinite admissible list l_s of its sequents $[S(N)[\theta^c\epsilon\cdots], S(N^h)[\theta_1^c\tau\cdots], S(N^{h'})[\cdots], \ldots]$ and define the path p *underlying* l_s as the concatenation of the rb-paths from π that built s, i.e., $[N, \ldots, B, N^h, \ldots, N^{f'}, \ldots]$. By the construction of s, every successive $(Subst)$-, bud and root nodes in p are labelled by the same sequent instance in l_s, so the $(Subst)$-steps are stuttering in l_s. By Lemma 10, all (Gen)- can be replaced by (Wk)-steps. p is of the form $[N_\infty \ldots, N_1, \ldots, N_0]$ where N_0, N_1, $\ldots$, N_∞ are an infinite number of *all* the occurrences of N in p.

We will show that there is a trace following p that has an infinite number of progress points. As explained in [Brotherston and Simpson, 2011], it means that there is an infinite strictly decreasing sequence of ordinals, hence contradiction. Since p is the concatenation of rb-paths in π and π discharges its IHs, for each such rb-path the bud sequent is $<_\pi$-derivable from the cumulative instance, along the rb-path, of the root sequent. By Lemma 11, there is an instance $S(N_\infty)[\theta_\infty]$ such that $S(N_0)$ is $<_\pi$-derivable from it along p, where θ_∞ is the composition of all cumulative substitutions of the rb-paths from l. For any two consecutive nodes N_i and N_{i-1} ($i \in [1..\infty]$), we have that $S(N_{i-1})[\theta_{i-1}]$ is $<_\pi$-derivable from $S(N_i)[\theta_i]$, where θ_i (resp., θ_{i-1}) are the compositions of all cumulative substitutions of the rb-paths along $[N_i, \ldots, N_0]$ (resp., $[N_{i-1}, \ldots, N_0]$).

Let us denote by S (resp, S') the sequent $S(N_i)[\theta_i]$ (resp., $S(N_{i-1})[\theta_{i-1}]$), for some $i \in [1..\infty]$. By Definition 11 and the transitivity of the $<_\pi$-derivability relation, for each IAA l from A_S there is an IAA l' from $A_{S'}$ such that l derives from l'. Therefore, there are n traces along the path p' $[N_\infty, \ldots, N_i]$, where n is the number of IAAs from S.

We will show that the traces along p' have an infinite number of progress points. By contradiction, we assume that this number is finite. Therefore, there is a subpath p'' of p whose traces have no progress points and there exists $j \in [1..\infty]$ such that N_j and N_{j-1} belong to p''. Let us denote by S_j (resp, S_{j-1}) the sequent $S(N_j)[\theta_j]$ (resp., $S(N_{j-1})[\theta_{j-1}]$). Since S_{j-1} is $<_\pi$-derivable from S_j, we have that $A_{S_{j-1}} <_\pi A_{S_j}$. By the definition of $<_\pi$ as a multiset extension of the ordering $<_a$ over the instances of IAAs from the root sequents in π, there should be an IAA $l \in A_{S_{j-1}}$ for which there is another IAA $l \in A_{S_j}$ such that $l <_a l'$ and l derives from l', i.e., l and l' are from an infinite trace t following (a subpath of) p'' which has no progress points. According to the definition of a trace (see Definition 8) and the way l_s was built, $l <_a l'$ is possible only if the subtrace of t from l' to l has at least one progress point, so contradiction. Otherwise, $l \equiv l'$ since i) l_s is admissible, ii) the $(Subst)$-steps are stuttering, iii) the (Gen)-steps can be replaced by (Wk)-steps, and iv) the instantiation steps that built s preserve the equality relations. □

Example 26 *For the sequents labelling the nodes from the digraph given in Example 24, we define the measure values $A_{Nt\vdash R(t,0)} = \{Nt\}$, $\forall t$, and $A_{Nt_1,Nt_2\vdash R(t_1,t_2)} = \{Nt_2\}$, $\forall t_1, t_2$. The IH $S(N^{13})$ is $<_\pi$-derivable from $S(N^{10})[\{x' \mapsto sx''\}]$, hence discharged by the SCC π using the trace $[Nx', Nx'', Nx'']$, if $\{Nx''\} <_\pi \{Nsx''\}$. Also, the IH $S(N^7)$ is $<_{\pi'}$-derivable from $S(N^1)[\{x \mapsto sx'; y \mapsto sy'\}]$ in the SCC π' using the trace $[Ny, Ny, Ny', Ny', Ny', Ny']$ if $\{Ny'\} <_{\pi'} \{Nsy'\}$. The ordering constraints hold if $<_\pi$ and $<_{\pi'}$ are the multiset extensions of a rpo $<_{rpo}$ for which $z <_{rpo} sz$, for every variable z.*

By Theorem 19, the root sequents in the pre-proof tree-set, $S(N^1)$ and $S(N^{10})$, are true.

Validation costs. We analyse the worst-case time complexity for validating the soundness of a pre-proof tree of n nodes with p ($< n$) buds occurring in non-singleton SCCs. The number of operations for normalising a CLKID$^\omega_N$ pre-proof of n nodes is given by the sum of non-root companions, non-terminal $(Subst)$-nodes and nodes labelled by some sequent that is the premise of a rule r different from $(Subst)$. So, it is smaller than $3n$. Let c be the maximal cost of an operation, including the node duplication and the creation of a $(Subst)$-node or bud-companion relation. Their total cost is smaller than $4nc$ (the second operation duplicates nodes twice). If c' is the constant representing the cost for annotating a

substitution, the cost for building the digraph of the normalized pre-proof tree-set is smaller than nc'. The partition of a digraph in SCCs can be done in linear time [Tarjan, 1972].

The cost for evaluating the ordering-derivability constraints is computed as follows. If B denotes a bud occurring in a non-singleton SCC, the IH that instantiates $S(B)$ is unique because B has only one companion and at most one companion can be the root of the tree including B. The number of $<$-derivability constraints is that of their buds, i.e., p. In the worst case, p is $n-1$. The validation cost of a $<$-derivability constraint is the sum of the costs of derivability and ordering constraints. The time complexity for checking whether an IAA l derives from another IAA l' is linear w.r.t. the size of the history of l, which is a value smaller than n. The time complexity for checking a multiset extension relation is $O(rq)$, where r and q are the number of IAAs from the measure value of the compared sequents. In the worst case, when all bud IAAs have their history of length n and p is $n-1$, the time complexity for checking the derivability constraints is $O(n^2k^2)$, where k is the maximal cardinality of a sequent's measure value. Similarly, the worst-case validation cost of the ordering constraints is polynomial in k, the maximal size of a literal and n, if the time complexity for comparing two IAAs is at most polynomial, for example by using a Knuth-Bendix ordering [Baader and Nipkow, 1998].

7.3 Implementation

The method has been implemented in CYCLIST and its extended version is called E-CYCLIST. CYCLIST builds the pre-proofs using a depth-first search strategy that aims at closing open nodes as quickly as possible. Whenever a new cycle is built, the model checker is called to validate it. If the validation result is negative, the prover backtracks by trying to find another way to build new cycles. Hence, it may happen that the model checker be called several times during the construction of a pre-proof.

To each root r from the digraph $\mathcal{P}$ of a normalized pre-proof tree-set, the method attaches a measure $\mathcal{M}(r)$ consisting of a multiset of IAAs of the sequent labelling r, denoted by $S(r)$. The procedure for computing these measure values is given by Algorithm 3.

Algorithm 3 GenOrd($\mathcal{P}$): to each root r of $\mathcal{P}$ is attached a measure $\mathcal{M}(r)$

```
for all root r do
  M(r) := ∅
end for
for all rb-path r → b from a non-singleton SSC do
  if there is a trace between an IAA A of S(b) and an IAA A' of S(r) then
    add A to M(rc) and A' to M(r), where rc is the companion of b
  end if
end for
```

Since the number of rb-paths is finite, Algorithm 3 terminates.

Example 27 *The measure values for the roots from the normalised pre-proof tree-set from Example 24 are built as follows.*

Firstly, the measure for the roots, denoted by $(*)$ *and* $(\dagger)$*, are the empty multisets. Then, we analyse each rb-path found in a non-singleton SCC of the digraph from Example 25. The rb-path from* $(\dagger)$ *to the bud* $(\dagger)$ *has a trace* Nx', Nx'', Nx'', Nx'*, so the new measure value for* $(\dagger)$ *is* $\{Nx'\}$*. Finally, the rb-path from* $(*)$ *to the bud* $(*)$ *has a trace* $Ny, Ny, Ny', Ny', Ny', Ny$*, so the new measure value for* $(*)$ *is* $\{Ny\}$*.*

Our method is *semi-decidable* as Algorithm 3 may compute measure values that do not pass the comparison test for some non-singleton SCCs that are validated by the model checker. For this case, we considered an *improvement* consisting of the incremental addition of IAAs from a root sequent that are not in the measure value of the corresponding root r. Such an addition does not affect the comparison value along the rb-paths starting from r, it affects only the comparison tests for the rb-paths ending in the companions of r. This may duplicate some IAAs from the value measure of the roots from the rb-paths leading to these companions and the duplicated IAAs have to be processed as the added IAAs from the beginning. And so on, until no changes are performed anymore.

Table 7.1 illustrates some statistics about the proofs of the conjectures considered in Table 1 from [Brotherston *et al.*, 2012]. They have been produced with CLKID$^{\omega}_{N}$ by using or not our improved method integrated in a version of CYCLIST tagged as 'CSL-LICS14' in the git repository at https://github.com/ngorogiannis/cyclist. The IAAs are indexed in CYCLIST to facilitate the construction of traces and the way they are indexed influence how the proofs are built. The column labelled 'Time-E' (resp., 'Time') is the proof time measured in milliseconds (resp. not) using our method. 'SC%' shows the percentage of time taken to check soundness. 'Depth' shows the depth of the proof, 'Nodes' the number of nodes in the proof, and 'Bckl.' the number of back-links in the proof. The last column shows the number of calls to the model checker as (calls on unsound proof)/(total calls) when our method is not used. The proofs have been performed on a MacBook Pro with a 2,7 GHz Intel Core i7 processor and 16 GB of memory. We can notice that, by using our method, the execution time is reduced by a factor going from 1.43 to 5.

Theorem	Time-E	Time	SC%	Depth	Nodes	Bckl.	Uns./All
$O_1x \vdash Nx$	2	7	61	2	9	1	0/1
$E_1x \vee O_2x \vdash Nx$	4	11	63	3	19	2	0/4
$E_1x \vee O_1x \vdash Nx$	2	9	77	2	13	2	2/5
$N_1x \vdash Ox \vee Ex$	3	7	52	2	8	1	0/1
$N_1x \wedge N_2y \vdash Q(x,y)$	297	425	40	4	19	3	168/181
$N_1x \vdash Add(x,0,x)$	1	5	76	1	7	1	0/1
$N_1x \wedge N_2y \wedge Add_3(x,y,z) \vdash Nz$	8	14	38	2	8	1	4/5
$N_1x \wedge N_2y \wedge Add_3(x,y,z) \vdash Add(x,sy,sz)$	15	22	32	2	14	1	9/10
$N_1x \wedge N_2y \vdash R(x,y)$	266	484	48	4	35	5	149/170

Table 7.1: Statistics for proofs using CYCLIST.

Even with the improved version of Algorithm 3, the method remains semi-decidable.

Example 28 *Figure 7.5 is the screen capture of a* CYCLIST *derivation built while proving* $Nx \wedge Ny \vdash R(x,y)$. *It can be noticed that that the computed measure values did not pass the comparison test while the model checker succeeds. Luckily, the prover backtracked and finally found the same proof as that built using the model checker.*

The source code of the implementation can be downloaded from https://members.loria.fr/SStratulat/files/e-cyclist.zip.

7.4 Converting cyclic to Noetherian induction reasoning

Cyclic pre-proofs can also be validated by Noetherian induction reasoning. The conversion process is explained using as example the CYCLIST proof of the P&Q example.

7.4.1 The CYCLIST proof

The inductive predicates P, Q and N from the P&Q example are encoded in CYCLIST as :

$$
\begin{array}{lll}
N\ \{ & P\ \{ & Q\ \{ \\
\quad true \Rightarrow N(0) & \quad true \Rightarrow P(0) & \quad true \Rightarrow Q(x,0) \\
\quad N_1(x) \Rightarrow N(s(x)) & \quad P_1(x)\ \&\ Q_2(x,s(x)) \Rightarrow P(s(x)) & \quad Q_1(x,y)\ \&\ P_2(x) \Rightarrow Q(x,s(y)) \\
\};\ & \};\ & \};
\end{array}
$$

The CYCLIST proof of the P&Q conjecture from Example 2 is given in Figure 7.6.

```
0: N_1(x) /\ N_2(y) |- R_1(x,y) (N L.Unf.) [1,2]
  1: N_2(y) /\ N_3(0) |- R_1(0,y) (R R.Unf.) [3]
    3: N_2(y) /\ N_3(0) |- T (Id)
  2: N_1(z) /\ N_2(y) /\ N_3(s(z)) |- R_1(s(z),y) (N L.Unf.) [4,5]
    4: N_1(z) /\ N_3(s(z)) /\ N_4(0) |- R_1(s(z),0) (R R.Unf.) [6]
      6: N_1(z) /\ N_3(s(z)) /\ N_4(0) |- R_1(z,0) (Weaken) [8]
        8: N_1(z) /\ N_2(0) |- R_1(z,0) (Subst) [9]
          9: N_1(x) /\ N_2(y) |- R_1(x,y) (Backl) [0]
    5: N_1(z) /\ N_2(w) /\ N_3(s(z)) /\ N_4(s(w)) |- R_1(s(z),s(w)) (R R.Unf.) [7]
      7: N_1(z) /\ N_2(w) /\ N_3(s(z)) /\ N_4(s(w)) |- R_1(s(s(z)),w) (N L.Unf.) [10,11]
        10: N_1(z) /\ N_3(s(z)) /\ N_4(s(0)) /\ N_5(0) |- R_1(s(s(z)),0) (R R.Unf.) [12]
          12: N_1(z) /\ N_3(s(z)) /\ N_4(s(0)) /\ N_5(0) |- R_1(s(z),0) (Weaken) [14]
            14: N_1(z) /\ N_3(s(z)) /\ N_4(0) |- R_1(s(z),0) (Backl) [4]
        11: N_1(z) /\ N_2(y) /\ N_3(s(z)) /\ N_4(s(s(y))) /\ N_5(s(y)) |- R_1(s(s(z)),s(y)) (R R.Unf.) [13]
          13: N_1(z) /\ N_2(y) /\ N_3(s(z)) /\ N_4(s(s(y))) /\ N_5(s(y)) |- R_1(s(s(s(z))),y) (N L.Unf.) [15,16]
            15: N_1(z) /\ N_3(s(z)) /\ N_4(s(s(0))) /\ N_5(s(0)) /\ N_6(0) |- R_1(s(s(s(z))),0) (R R.Unf.) [17]
              17: N_1(z) /\ N_3(s(z)) /\ N_4(s(s(0))) /\ N_5(s(0)) /\ N_6(0) |- R_1(s(s(z)),0) (Weaken) [19]
                19: N_1(z) /\ N_3(s(z)) /\ N_4(s(0)) /\ N_5(0) |- R_1(s(s(z)),0) (Backl) [10]
            16: N_1(z) /\ N_2(w) /\ N_3(s(z)) /\ N_4(s(s(s(w)))) /\ N_5(s(s(w))) /\ N_6(s(w)) |- R_1(s(s(s(z))),s(w)) (R R.Unf.) [18]
              18: N_1(z) /\ N_2(w) /\ N_3(s(z)) /\ N_4(s(s(s(w)))) /\ N_5(s(s(w))) /\ N_6(s(w)) |- R_1(s(s(s(s(z)))),w) (Open)

Miss !!!
Root list: 4, 10, 0

Measures proposed for the roots in cycles:
    4: [1, 1, 1, 3, 1]
    10: [3, 1, 3, 1, 1, 3, 4, 1]
    0: [1, 1, 1, 2, 1]
Checking the link of IAAs from buds to roots:
    10 to 0: | 1 -> 1 [true ]| 3 -> 1 [false ]| 4 -> 2 [false ] ==> true
    19 to 0: | 1 -> 1 [true ]| 3 -> 1 [false ]| 4 -> 2 [true ] ==> true
    4 to 0: | 1 -> 1 [true ]| 3 -> 1 [false ] ==> true
    14 to 10: | 1 -> 1 [false ]| 3 -> 3 [false ]| 4 -> 5 [false ] ==> true
    9 to 4: | 1 -> 1 [false ]| 2 -> 4 [false ] ==> false
The proof has NOT succeeded
Checking soundness starts...
Checking soundness ends, result=OK
```

Figure 7.5: A case when the comparison tests fail while the model checker succeeds.

Most of the inference rules are similar to those used to build the pre-proof from the previous section. $(L.Unf.)$ implements the CLKID$^\omega_N$'s $(Case)$-rule; it is prefixed by the inductive predicate symbol that has served to generate the variable instantiation schema. A case analysis is performed instead when the argument of the inductive predicate is a non-variable term (see its application on the sequent labelling the node [12]). $(R.Unf.)$ is similar to the CLKID$^\omega_N$'s $(R.)$-rule, excepting that it is less precise since only the used inductive predicate symbol is specified instead of the exact axiom. It also allows to unfold inside the conjunctions from the succedent part of the sequents. $(R.And)$ and $(Weaken)$ correspond to $(\wedge\ R)$ and (Wk), respectively. Other rules are:

- (Id), applied on leaves labelled with sequents whose succedent is known to be true,
- $(Backl)$, applied on every bud; it is followed by a singleton list containing the companion,
- $(Subst)$, the LK's substitution rule, and
- $(Ex\ Falso)$, applied on leaves labelled with sequents whose antecedent is false.

The last rule assumes that the function symbols are distinct. In addition, Cyclist assumes that they are injective.

Compared to the original pre-proof, the cyclic pre-proof is enriched with the bud-companion relations. The trace-based soundness arguments showing that it is indeed a proof are not given here. Instead, the reader may consult Example 5.2.3 from [Brotherston, 2006] presenting a different CLKID$^\omega$ proof of our P&Q conjecture.

7.4.2 The conversion procedure

Our conversion procedure is based on the ordering-based checking procedure.

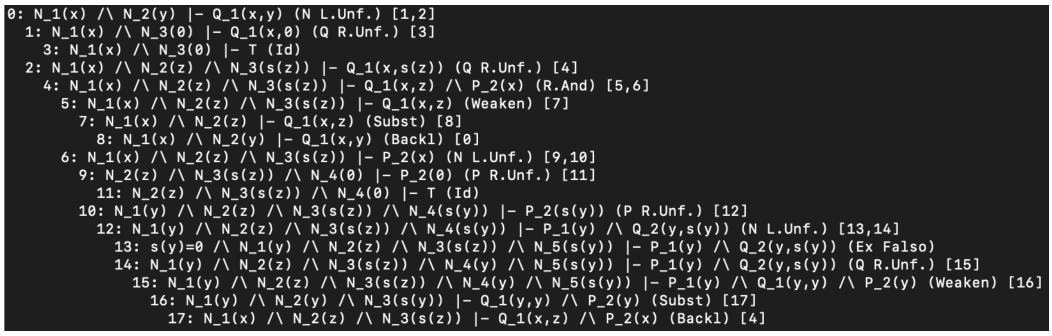


Figure 7.6: The screenshot of the Cyclist pre-proof built for the P&Q conjecture.

We recall that, due to the bud-companion relations, the Cyclist pre-proof trees can be represented as graphs that may have non-singleton SCCs, hence allowing for building infinite paths along their cycles. The checking procedure can normalize any Cyclist pre-proof tree $\mathcal{T}$, having only $(L.Unf.)$-steps that instantiate variables along the cycles, into a forest of connected cyclic pre-proof trees for which i) all companions are root nodes, ii) the root of $\mathcal{T}$ is among the root nodes, and iii) every rb-path p, also denoted by $r \to b$ when the root r leads to the bud b, ends with a $(Subst)$-step which is also the unique $(Subst)$-step in p.

For the P&Q example, the result of the normalisation operation generates two trees, where the node 4 is duplicated twice. Once, to separate from the main tree the subtree rooted by it, followed by the creation of a new bud-companion relation between the nodes 4 and $4''$. The other time, to perform a stuttering step, by applying a $(Subst)$-step with the identity substitution. Below is listed the result of the normalisation step, where only the rb-paths are displayed:

$$\dfrac{\dots \qquad \dfrac{\dfrac{\dfrac{(Backl)[4]}{4'' : N_1x, N_2z, N_3sz \vdash Qxz \wedge Px}\,(Subst)[4'']}{4' : N_1x, N_2z, N_3sz \vdash Qxz \wedge Px}\,(Q\ R.Unf.)[4']}{2 : N_1x, N_2z, N_3sz \vdash Qxsz}\,(N\ L.Unf.)[1,2]}{0 : N_1x, N_2y \vdash Qxy}$$

$$\dfrac{\dfrac{\dfrac{(Backl)[0]}{8 : N_1x, N_2y \vdash Qxy}\,(Subst)[8]}{7 : N_1x, N_2z \vdash Qxz}\,(Weaken)[7] \quad \dots \quad \dfrac{\dfrac{\dfrac{\dfrac{(Backl)[4]}{17 : N_1x, N_2z, N_3sz \vdash Qxz \wedge Px}\,(Subst)[17]}{16 : N_1y, N_2y, N_3sy \vdash Qyy \wedge Py}\,(Weaken)[16]}{15 : N_1y, N_2z, N_3sz, N_4y, N_5sy \vdash Py \wedge Qyy \wedge Py}\,(Q\ R.Unf.)[15]}{14 : N_1y, N_2z, N_3sz, N_4y, N_5sy \vdash Py \wedge Qysy}\,(N\ L.Unf.)[13,14]}{12 : N_1y, N_2z, N_3sz, N_4sy \vdash Py \wedge Qysy}\,(P\ R.Unf.)[12]}{10 : N_1y, N_2z, N_3sz, N_4sy \vdash Psy}\,(N\ L.Unf.)[9,10]}{6 : N_1x, N_2z, N_3sz \vdash Px}\,(R.And)\ [5,6]}{4 : N_1x, N_2z, N_3sz \vdash Qxz \wedge Px}$$

It can be noticed that every infinite path in the Cyclist pre-proof are also reproduced in its normal form, by duplicating the node 4, and viceversa, by deleting some of its occurrences.

The next step is to associate to each root r a measure value in terms of a multiset of inductive atoms occurring in the antecedent part of the sequent labelling r such that some derivability and ordering constraints are satisfied. The derivability constraints consist in checking that, given an rb-path p, some IAA of the sequent labelling its bud can be derived from some IAA of the sequent labelling its root, by following a trace along p. For our example, we list all traces along the rb-path

- $0 \to 4''$: $[N_1x, N_1x, N_1x, N_1x](=)$, $[N_2y, N_2z, N_2z, N_2z](>)$, $[N_2y, N_3sz, N_3sz, N_3sz](=)$,
- $4 \to 8$: $[N_1x, N_1x, N_1x, N_1x](=)$, $[N_2z, N_2z, N_2z, N_2y](=)$, and
- $4 \to 17$: $[N_1x, N_1x, N_1y, N_1y, N_1y, N_1y, N_1x](>)$, $[N_1x, N_1x, N_4sy, N_4sy, N_4y, N_4y, N_4y, N_2z](>)$, and $[N_1x, N_1x, N_4sy, N_4sy, N_5sy, N_5sy, N_3sy, N_3sz](=)$.

Each trace $[A_r, ..., A_b]$ following any of the above rb-paths was annotated by $>$ if it has at least one *progress point*, i.e., a step where an IAA changes due to an unfolding occurring inside the application of $(L.Unf.)$, fact also denoted by $A_r > A_b$. Otherwise, it is annotated by $=$ and say that A_r and A_b are *equal*. An infinitely progressing trace has infinite progress points.

The global trace condition requires that, for any infinite path p, there is an infinitely progressing trace starting from some point of p. This condition is ensured if for each rb-path $r \to b$ of p, we have $M(r) >_{mul} M(c(b))$, where $c(b)$ is the companion of b and $M(r)$ (resp., $M(c(b))$) are the measure values for r (resp., $c(b)$). $>_{mul}$ is the *trace-based multiset extension* of $>$, i.e., $M(r) >_{mul} M(c(b))$ if the equal IAAs from $M(r)$ and $M(c(b))$ are deleted to get $M(r)'$ and $M(c(b))'$ and either i) $M(c(b))'$ is empty and $M(r)'$ is not empty, or ii) for each IAA $a \in M(c(b))'$ there is an IAA $b \in M(r)'$ such that $b > a$.

The conversion procedure was implemented in E-Cyclist. For our example, the heuristics proposed the measure value $\{N_1x, N_1x, N_2y, N_2y\}$ for the node 0 and $\{N_1x, N_1x, N_2z, N_2z, N_3sz\}$ for the node 4. The derivability and ordering constraints hold for each rb-path, as follows:

- the rb-path $0 \to 4''$: $\{N_1x, N_1x, N_2y, N_2y\} >_{mul} \{N_1x, N_1x, N_2z, N_2z, N_3sz\}$. The deletion of equal IAAs gives $\{N_2y\} >_{mul} \{N_2z, N_2z\}$ which holds since $N_2y > N_2z$;
- the rb-path $4 \to 8$: $\{N_1x, N_1x, N_2z, N_2z, N_3sz\} >_{mul} \{N_1x, N_1x, N_2y, N_2y\}$. It boils down to $\{N_3sz\} >_{mul} \{\}$;
- the rb-path $4 \to 17$: $\{N_1x, N_1x, N_2z, N_2z, N_3sz\} >_{mul} \{N_1x, N_1x, N_2z, N_2z, N_3sz\}$. The equal IAAs are deleted to issue $\{N_1x, N_2z, N_2z, N_3sz\} >_{mul} \{N_1x, N_1x, N_2z, N_2z\}$, for which N_1x found on the lhs of $>_{mul}$ is greater than any IAA from $\{N_1x, N_1x, N_2z, N_2z\}$.

The measure values built for the roots help also to compare the roots using well-founded orderings. Given a well-founded ordering $<^{wf}$ over the IAAs and $<^{wf}_{mul}$ its multiset extension, we say that an rb-path $r \to b$ is *valid* if $M(c(b))[\delta] <^{wf}_{mul} M(r)[\theta]$, where δ is the substitution used in the *Subst*-step of $r \to b$ and θ is the *cumulative substitution* for $r \to b$, i.e., the composition of all substitutions instantiating IAAs along $r \to b$. In the Noetherian induction setting, $M(c(b))[\delta]$ plays the role of *induction hypothesis* and $M(r)[\theta]$ that of the *induction conclusion*. We recall that σ_{id} denotes the identity substitution.

Showing that every rb-path is valid is enough to conclude that a pre-proof is a Noetherian induction proof. For our example, the ordering constraints checking the validity of the rb-paths are:

- $0 \to 4''$: $\{Nx, Nx, Nz, Nz, Nsz\}[\sigma_{id}] <^{wf}_{mul} \{Nx, Nx, Ny, Ny\}[\{y \mapsto sz\}]$,
- $4 \to 8$: $\{Nx, Nx, Ny, Ny\}[\{y \mapsto z\}] <^{wf}_{mul} \{Nx, Nx, Nz, Nz, Nsz\}[\sigma_{id}]$, and
- $4 \to 17$: $\{Nx, Nx, Nz, Nz, Nsz\}[\{x \mapsto y; z \mapsto y\}] <^{wf}_{mul} \{Nx, Nx, Nz, Nz, Nsz\}[\{x \mapsto sy\}]$, where the indexes for the inductive predicates are omitted.

The ordering constraints are satisfied if $<^{wf}$ is a mpo built from any precedence over the symbols $\{N, s, 0\}$. In practice, one can consider that all the inductive predicate symbols are equivalent, hence the measure values for the roots are multisets built only from their arguments. For our example, the measure value for the root 0 would be $\{x, x, y, y\}$ and $\{x, x, z, z, sz\}$ for the root 4.

7.4.3 Experimental results

We report that the proofs for all these conjectures have been successfully converted to Noetherian induction proofs. Some statistics about the Noetherian induction proofs, as the number of SCCs, the root sequents and their measure values, are given in Table 7.2:

Theorem	# SCC	Root Sequents	Measure values
$O_1(x) \vdash N(x)$	1	$O_1(x) \vdash N(x)$	$\{x\}$
$E_1(x) \vee O_2(x) \vdash N(x)$	2	$E_1(x) \vdash N(x)$	$\{x\}$
		$O_2(x) \vdash N(x)$	$\{x\}$
$E_1(x) \vee O_1(x) \vdash N(x)$	1	$E_1(x) \vdash N(x)$	$\{x\}$
		$O_1(x) \vdash N(x)$	$\{x\}$
$N_1(x) \vdash O(x) \vee E(x)$	1	$N_1(x) \vdash O(x) \vee E(x)$	$\{x\}$
$N_1(x) \wedge N_2(y) \vdash Q(x,y)$	1	$N_1(x) \wedge N_2(y) \vdash Q(x,y)$	$\{x,x,y,y\}$
		$N_1(x) \wedge N_2(z) \wedge N_3(s(z)) \vdash Q(x,z) \wedge P(x)$	$\{x,x,z,z,s(z)\}$
$\vdash Add(0,0,0)$	0		
$N_1(x) \vdash Add(x,0,x)$	1	$N_1(x) \vdash Add(x,0,x)$	$\{x\}$
$N_1(x) \wedge N_2(y) \wedge Add_3(x,y,z) \vdash N(z)$	1	$N_1(x) \wedge N_2(y) \wedge Add_3(x,y,z) \vdash N(z)$	$\{x,z\}$
$N_1(x) \wedge N_2(y) \wedge Add_3(x,y,z) \vdash Add(x,s(y),s(z))$	1	$N_1(x) \wedge N_2(y) \wedge Add_3(x,y,z) \vdash Add(x,s(y),s(z))$	$\{x,z\}$
$N_1(x) \wedge N_2(y) \vdash R(x,y)$	2	$N_1(x) \wedge N_2(y) \vdash R(x,y)$	$\{y\}$
		$N_1(w) \wedge N_3(s(s(w))) \wedge N_4(s(w)) \wedge N_5(0) \vdash R(s(w),0)$	$\{w\}$

Table 7.2: Statistics about the Noetherian induction proofs issued from CYCLIST proofs.

7.5 Conclusions

We have presented a new method to validate a class of CLKID$^\omega$ pre-proof trees by converting them to pre-proof tree-sets, then showing that the global trace condition is implicitly satisfied if some ordering and derivability constraints hold. Every infinite path p from a pre-proof tree normalized to a proof (tree-set) can be built by concatenating path segments from the definition of the minimal cycles of its proof. These constraints ensure that there is an infinitely progressing trace following some tail of p. The ordering constraints can also be used to validate cyclic pre-proofs as Noetherian induction proofs.

Our approach allows more flexibility; a different induction ordering can be defined for each SCC with cycles from the digraph of the proof. This is not the case from the unique induction ordering defined over the buds of a pre-proof tree with trace manifolds [Brotherston, 2005, Brotherston, 2006]. Also, our approach does not require pre-proof trees to be in cycle normal form that are, in the worst case, exponentially bigger.

The soundness check can be done in polynomial time provided that the time complexity for comparing two IAAs is at most polynomial. We defined proof strategies ensuring that the number of ordering constraints equals that of the induction hypotheses that are really required in the proof. In practice, their number is generally small even for proofs concerning real-size applications. For example, every cyclic induction proof from Table 2.1 includes at most 8 induction hypotheses and 4 minimal cycles.

8

Saturation-based Reasoning

Sommaire

8.1 The logical framework 108
8.1.1 Contextual cover sets (CCSs) 108
8.1.2 The A^1 and A^2 inference systems 109
8.1.3 Reasoning modules 110
8.1.4 Methodology for designing and analysing CCS-based inference systems . . 110
8.1.5 Case study: designing an implicit induction inference system 111
8.2 Analysing and extending saturation-based inference systems 113
8.2.1 Case study 1: a paramodulation-based inference system 113
8.2.2 Case study 2: a resolution-based inference system 115
8.3 Conclusions 119

This chapter is based on [Stratulat, 2005, Stratulat, 2007]. We present a framework and a methodology to build and analyse formula-based Noetherian induction inference systems. A stronger connection between different proof techniques like those based on implicit induction and saturation is established by uniformly and explicitly representing them as applications of the formula-based Noetherian induction. The framework offers a clear separation between logic and computation, by the means of i) an *abstract inference system* that defines the maximal sets of induction hypotheses available at every step of a proof, and ii) *reasoning modules* that perform the computation and allow for modular design of the concrete inference rules. The methodology is applied to define a concrete implicit induction inferece system and two saturation-based inference systems.

In Chapter 1, we have shown that *soundness* is a vital property of formula-based Noetherian induction inference systems, which guarantees the persistence of minimal counterexamples in any derivation containing false conjectures. One straightforward application of it concerns the *implicit induction* inference systems which can prove that the conjectures are valid if the derivations end with an *empty* set. Another outstanding application refers to *saturation-based* inference systems; they detect the minimal counterexamples in the last (saturated) state in finite derivations.

The methodology helps to analyse two existent saturation-based inference systems, one based on ordered paramodulation, the other on ordered resolution.

Contributions. Below we listed the main contributions of our approach:

Consolidation of the link between saturation and implicit induction procedures. The design of inference systems that (partially) abstract the computation goes back to the late 80's, early 90's [Bachmair, 1988, Reddy, 1990], as a solution for representing classes of concrete inference systems sharing similar reasoning techniques. In this way, the properties established for abstract systems are naturally propagated to the instantiating systems. In our framework, we pushed this idea further and

proposed A_s, a saturation-based system that completely abstracts the computation, and showed the resolution-based system RP [Bachmair and Ganzinger, 2001] to be an instance of it. Since A_s defines the same induction hypotheses as another implicit induction abstract system [Stratulat, 2005], an even stronger link between saturation-based and implicit induction procedures has been established. This result opens the perspective of designing hybrid inference systems that would be able to perform both implicit induction and saturation-based proofs with minor changes, as explained in [Stratulat, 2005]. It also means that reasoning techniques can be shared: for example, we expect that the approach for integrating Computer Algebra algorithms into implicit induction systems [Armando *et al.*, 2002] be easily adaptable for saturation-based systems.

Definition of stronger redundancy criteria for saturation-based systems. Since similar instantiation results between concrete and abstract systems have already been obtained in [Stratulat, 2001, Stratulat, 2005], A_s can be considered as a reference for defining maximal sets of induction hypotheses for many other saturation-based systems. The proposed methodology leaves room for further extensions of existing systems, for example for any concrete RP instance described in [Bachmair and Ganzinger, 2001].

Simple and modular design of sound and concrete saturation-based systems. Prover designers can benefit of our framework to build new and sound saturation-based systems or to extend existing ones. Thanks to the proposed methodology, any rule can be soundly i) designed to integrate and tune specific reasoning techniques, by simply showing it as an instance of an A_s-rule, and/or ii) extended to use maximal sets of induction hypotheses. In our approach, the soundness proofs are modular; tedious and error-prone soundness proofs are therefore avoided even for very complex inference systems. The methodology also ensures that the extended systems preserve their refutational completeness, as we have shown for variants of RP. However, it cannot serve to establish the refutational completeness of arbitrary system (see later in the chapter that Property 2 is generally difficult to be proved). Therefore, our methodology completes rather than competes with other techniques for proving the refutational completeness property.

Structure of the chapter. The chapter is organised in 3 sections. Section 8.1 presents a framework and a methodology to design and analyse formula-based Noetherian induction inference systems. As a first application, the framework and methodology are used for building a very simple implicit induction inference system that is sound by construction. The framework and methodology are later applied in Section 8.2 to analyse and extend two saturation-based inference systems that use the paramodulation and resolution techniques. The last section concludes.

8.1 The logical framework

8.1.1 Contextual cover sets (CCSs)

We consider sound formula-based Noetherian induction inference systems consisting of sets of inference rules that guarantee the persistence of minimal counterexamples in any derivation, i.e. whenever the processed conjecture has a minimal counterexample, an equivalent counterexample exists in a future state. For example, if $\phi\tau$ is a minimal counterexample of ϕ at the step $E \cup \{\phi\} \vdash E \cup \Phi$, the completely automated inference systems require that a counterexample equivalent to $\phi\tau$ be in the next state $E \cup \Phi$. More precisely, the condition $\Gamma \models \phi\tau$ should be satisfied, where Γ can contain both $(E \cup \Phi)_{\sim\phi\tau}$ and true formulas: i) the axioms Ax, ii) $\mathsf{C}^1_{\leq\phi\tau}$, where C^1 has elements of E (but not of Φ) and other conjectures with no minimal counterexamples, iii) $\mathsf{C}^2_{<\phi\tau}$, where C^2 can be any set of conjectures from the derivation, and iv) $\Phi_{\leq\phi\tau}$. It is assumed that $\leq$ is a Noetherian quasi-ordering over formulas which is stable under substitutions.

Usually, the minimal counterexample $\phi\tau$ is hard to identify among the other ground instances of ϕ. To be sure that this condition is satisfied by $\phi\tau$, it is sufficient to generalise it for any ground instance of ϕ. To sum up, the condition becomes

$$Ax \cup \mathsf{C}^1_{\leq\phi\gamma} \cup \mathsf{C}^2_{<\phi\gamma} \cup \Phi_{\leq\phi\gamma} \models \phi\gamma, \text{ for any ground instance } \phi\gamma, \tag{8.1}$$

i.e. Φ is a (general) *contextual cover set* (CCS) of ϕ in the *context* $\mathsf{C} = (\mathsf{C}^1, \mathsf{C}^2)$, a concept also defined in [Stratulat, 2000a, Stratulat, 2000b, Stratulat, 2001].

Several kinds of CCSs are distinguished: i) *cover set* if $\mathsf{C}^1 = \mathsf{C}^2 = \emptyset$, ii) *strict* if $\Phi_{\leq\phi\gamma}$ is replaced by $\Phi_{<\phi\gamma}$, iii) *empty* if $\Phi = \emptyset$, and iv) *universal* if $Ax = \emptyset$. They are not mutually exclusive; for example, any empty CCS is also strict. The notion of cover set as a particular CCS corresponds to that from [Bronsard *et al.*, 1996], which is a generalisation of [Reddy, 1990]. Variants of it can be found in different completion-based induction methods [Bachmair, 1988, Zhang *et al.*, 1988, Kounalis and Rusinowitch, 1990b, Kapur *et al.*, 1991]. In the definition (8.1), the formulas from $\mathsf{C}^1_{\leq\phi\gamma}$, $\mathsf{C}^2_{<\phi\gamma}$ and $\Phi_{\leq\phi\gamma}$ can be used to deduce $\phi\gamma$ even if they are not true or not yet proved to be true. They play the role of *induction hypotheses*.

In the rest of the chapter, the 'contextually cover' relation is generalised to sets of formulas: $\Psi \sqsubseteq_{\mathsf{C}} (\sqsubset_{\mathsf{C}}) \Phi$ iff Φ is a (strict) CCS of any $\phi \in \Psi$ in the context C.

Properties As shown in [Stratulat, 2001], the 'contextually covers' relation is a quasi-ordering: i) (reflexivity) $\Phi \sqsubseteq_{\mathsf{C}} \Phi$, for any set of formulas Φ, and ii) (transitivity) for any set of formulas Φ, Ψ and Γ, if $\Phi \sqsubseteq_{\mathsf{C}} \Psi$ and $\Psi \sqsubseteq_{\mathsf{C}} \Gamma$, then $\Phi \sqsubseteq_{\mathsf{C}} \Gamma$. Due to the transitivity property of $\sqsubseteq_{\mathsf{C}}$, new 'contextually cover' relations can be obtained by composition operations.

- *horizontal composition.* Given the chain of 'contextually cover' relations $\Phi_1 \sqsubseteq_{\mathsf{C}} \ldots \sqsubseteq_{\mathsf{C}} \Phi_i \sqsubseteq_{\mathsf{C}} \Phi_{i+1} \sqsubseteq_{\mathsf{C}} \ldots \Phi_n$, then i) $\Phi_i \sqsubseteq_{\mathsf{C}} \Phi_j$, for all $i, j \in [1..n]$ with $i \leq j$, and ii) if $\Phi_i \sqsubset_{\mathsf{C}} \Phi_{i+1}$ then $\Phi_k \sqsubset_{\mathsf{C}} \Phi_j$, for all $k \leq i$ and $j > i$.
- *vertical composition.* Given the set of formulas $\Phi = \{\phi_1, \ldots, \phi_n\}$ such that $\forall i \in [1..n]$, $\{\phi_i\} \sqsubseteq_{\mathsf{C}}$ (resp. $\sqsubset_{\mathsf{C}}$) Ψ_i then $\Phi \sqsubseteq_{\mathsf{C}}$ (resp. $\sqsubset_{\mathsf{C}}$) $\bigcup_{j=1}^{n} \Psi_j$.

Our framework is able to provide maximal sets of induction hypotheses at any step of a derivation by the means of two inference systems A^1 and A^2.

8.1.2 The A^1 and A^2 inference systems

1-AddPremise	1-Simplify
$(E \cup \{\phi\},\ H) \vdash^{\mathsf{A}^1} (E \cup \Phi,\ H \cup \{\phi\})$ if $\{\phi\} \sqsubset_{(H,\ E)} \Phi$	$(E \cup \{\phi\},\ H) \vdash^{\mathsf{A}^1} (E \cup \Phi,\ H)$ if $\{\phi\} \sqsubseteq_{(E \cup H,\ \emptyset)} \Phi$

Figure 8.1: The one-step inference rules.

Generally, the CCS contexts may have conjectures from the whole derivation to be computed at any inference step. To *automate* their computation, a new set of particular formulas will join the set of conjectures such that the context has only formulas from the current state of the derivation. They are called *premises* and represent processed conjectures that do not contain minimal ground formulas. The inference rules have now the form:

$$\textsc{Name} \quad (E \cup \{\phi\},\ \mathbf{H}) \vdash (E \cup \Phi,\ \mathbf{H}') \ [\text{if } \textit{Conditions}]$$

where $\mathbf{H}$ and $\mathbf{H}'$ are premises. The inference system A from Subsection 4.1.2 can be represented using CCSs as in Figure 8.1 by the A^1 inference system. Its rules replace ϕ by Φ if Φ is a CCS of ϕ, whose context has only formulas from E and H. In its simplest form, it consists of two inference rules that build the new set of conjectures in just one step. The 1-AddPremise rule firstly computes Φ as a strict CCS of ϕ, then adds it to the current set of premises to participate to further computations. 1-Simplify does not make such addition, but it is less restrictive: Φ can be a general CCS and instances of E equivalent to ϕ are allowed. The induction hypotheses from the contexts do not affect the soundness of A^1.

Theorem 20 *The minimal counterexamples are persistent in any A^1-derivation starting with an empty set of premises.*

Proof Similar to the proof of Theorem 6. □

2-AddPremise
$(E \cup \{\phi\}, H) \vdash^{\mathsf{A}^2} (E \cup \{\phi\} \cup \underbrace{\Phi_1 \cup \ldots \cup \Phi_p}_{\Phi}, H \cup \{\phi\})$ **if**
(a) $\{\phi\} \sqsubseteq_{(H, E \cup \Phi)} \bigcup_{j=1}^{p} \{\psi_j\}$ **and**
(b) $\{\psi_j\} \sqsubset_{(H, E \cup \{\phi\} \cup (\Phi \setminus \Phi_j))} \Phi_j$, for each $j \in [1..p]$

2-Simplify
$(E \cup \{\phi\}, H) \vdash^{\mathsf{A}^2} (E \cup \{\phi\} \cup \underbrace{\Phi_1 \cup \ldots \cup \Phi_p}_{\Phi}, H)$ **if**
(a) $\{\phi\} \sqsubseteq_{(E \cup H \cup \Phi, \emptyset)} \bigcup_{j=1}^{p} \{\psi_j\}$ **and**
(b) $\{\psi_j\} \sqsubseteq_{(E \cup H \cup (\Phi \setminus \Phi_j), \{\phi\})} \Phi_j$, for each $j \in [1..p]$

Figure 8.2: The two-step inference rules.

More complex inference rules can be designed by using the composition properties of CCSs. The inference rules of the inference system A^2 from Figure 8.2 build the new set of conjectures in two steps. At the step (a) of 2-AddPremise, an intermediate CCS of ϕ is created, denoted by Ψ. Then, for any formula from Ψ a strict CCS is built and added as new conjectures. By vertical composition at the step (b), Φ is a strict CCS of Ψ. Φ is also a strict CCS of ϕ, by horizontal composition. Similarly, 2-Simplify builds Φ as a CCS of ϕ. It can be shown that the two-step inference system is sound as in [Stratulat, 2001] and following the idea of the proof of Theorem 20. The one-step inference system is an instance of it by considering $\{\phi\}$ as the (trivial) CCS for $\{\phi\}$ at the step (a) of the corresponding two-step inference rules.

2-AddPremise is asymmetric in the construction of Φ. To be complete, the two-step inference system should include a variant of 2-AddPremise that creates the strict CCS at the step (a) instead of (b).

8.1.3 Reasoning modules

The inference systems A^1 and A^2 are abstract because they ignore how the CCSs from its inference rules are built. The last components of our framework, the *reasoning modules*, are in charge to compute the *elementary* CCSs, i.e. that are not built by composition operations. They represent implementations of reasoning techniques adequate to the nature of the employed consequence relation. The most part of them are deductive, like rewriting and subsumption, and can reason on any kind of consequence relation. Some others are more specific, as the replacement of a natural variable with 0, 1, ... for the initial consequences, or work only for particular reasoning domains, like the decision procedures.

A reasoning module is defined by two functions that take as arguments a context and a formula for which a CCS is built: i) the *generation* function g, and ii) the *condition* function *cond*, such that whenever $g(\phi, \mathsf{C}) = \Phi$ then $\{\phi\} \sqsubseteq_{\mathsf{C}} \Phi$ under the condition $cond(\phi, \mathsf{C})$. All the reasoning modules presented in the paper have the trivial condition function that returns true, but in general this is not the case. An inference system is *recursive* if the validation process of the conditions is performed by the prover itself. Establishing the soundness of recursive systems is more difficult because of the mutual dependency created between the prover and the reasoning modules: on the one hand, the inference rules need reasoning modules that soundly build CCSs and, on the other hand, the reasoning modules need sound validations of their conditions. An integration schema of reasoning modules in recursive implicit induction inference systems has been proposed in [Stratulat, 2001]. It has the advantage that elements of the context given as argument to the condition function can contribute as initial premises to the proof of the conditions.

8.1.4 Methodology for designing and analysing CCS-based inference systems

The framework can be used to design new inference systems and to analyse, improve and extend existing CCS-based inference systems. To build a new inference system that reasons on a given inductive consequence relation, the following steps can be followed:

1. provide a Noetherian ordering over the formulas that is stable under substitutions,

2. provide a set of reasoning techniques adequate for the consequence relation,
3. define reasoning modules based on the reasoning techniques,
4. instantiate abstract inference rules by showing how their CCSs are created by the reasoning modules.

	Kind of CCS produced by the generation function				
	general	strict	empty	cover set	universal
$\sqsubseteq$	true	true	true	true	true
$\sqsubset$	false	true	true	false	false

Table 8.1: The compatibility predicate.

The soundness of the new inference system is guaranteed if each inference rule is an instance of an A^1-rule by showing i) the *compatibility* and ii) the *context inclusion* of its CCSs w.r.t. the corresponding CCSs defined by the A^1-rule. The compatibility predicate is defined in Table 8.1. Any kind of generated CCS is compatible with general CCSs (third line) and only the strict and the empty CCSs are compatible with strict CCSs (last line). If the generation function builds a CCS of several kinds, for example a strict cover set, then the result is the disjunction of the compatibility results for each kind. A context $(\mathsf{C}^{11}, \mathsf{C}^{12})$ is *included* in $(\mathsf{C}^{21}, \mathsf{C}^{22})$ if $(\mathsf{C}^{11}_{\leq\phi} \cup \mathsf{C}^{12}_{<\phi}) \subseteq (\mathsf{C}^{21}_{\leq\phi} \cup \mathsf{C}^{22}_{<\phi})$, for any ground formula ϕ.

Analysing existing systems is an easier task since the ordering and the reasoning techniques are already provided. The most difficult steps are the representation of the derivation states under the form of (E, H) and that of the rules as instances of the abstract rules. Then, the rules can be soundly improved, for example, by expanding the contexts to the maximal values allowed by the abstract rules. Existing systems can be expanded modularly with new rules created in a flexible way, as described in the next section.

Theorem 21 ***(soundness and refutational completeness of variants)*** *Let $\mathcal{S}$ be a refutationally complete inference system that can be extended using the above methodology and let $\mathcal{S}'$ be such a variant. Then $\mathcal{S}'$ is sound and refutationally complete.*

8.1.5 Case study: designing an implicit induction inference system

An implicit induction inference system will be designed using our methodology to generate proofs that validate properties over the naturals from equational specifications.

Ordering The ordering over equalities $<_e$ can be defined as a multiset extension of a rpo $<_t$, starting from a precedence $<_{\mathsf{F}}$ over the function symbols, because any equality $s = t$ can be represented as the multiset $\{s, t\}$:

$$s = t \quad <_e \quad l = r \qquad \text{if } max(s,t) \ll_t max(l,r),$$

where $max(s,t)$ is the singleton containing the maximal term between s and t (wrt $<_t$) when it exists, otherwise $\{s,t\}$. We recall that $<_e$ and $<_t$ are Noetherian and stable under substitutions.

For example, given the precedence $0 <_{\mathsf{F}} S$ (the successor function) $<_{\mathsf{F}} +$, then i) $x <_t 0 + x$, ii) $S(x + y) <_t S(x) + y$ and iii) $S(x) + 0 = S(x) <_e S(S(x) + 0) = S(S(x))$.

Reasoning Techniques An inductive consequence relation appropriate for equalities is the initial consequence, denoted by $\models_{ini}$, and a typical reasoning technique for it is the replacement of a variable with finite descriptions of its domain. For the domain of the naturals, the most common are $\{0, S(x)\}$ and $\{0, S(0), S(S(x))\}$.

Two more general, deductive reasoning techniques, will be used: the elimination of identities of the form $s = s$ because they cannot contain counterexamples, and the rewriting. By $e[t]_p$ is unambiguously indicated that the equality (or term) e contains the term t at the position p. Then, given a set of rewrite rules ρ, the rewrite relation $\rightsquigarrow_\rho$ is defined as $e'[a'\sigma]'_p \rightsquigarrow_\rho e'[b'\sigma]'_p$ if $a' \to b' \in \rho$ and σ is a substitution.

Reasoning Modules The previously presented reasoning techniques will serve to build new reasoning modules:

- D. Its generation function is $g_D(t = t, (\emptyset, \emptyset)) = \emptyset$ and returns an empty CCS.
- R, with $g_R(e, (\mathsf{C}^1, \mathsf{C}^2)) = \{e'\}$, where $e \rightsquigarrow_{Ax \cup \mathsf{C}^1_{\leq e} \cup \mathsf{C}^2_{< e}} e'$. It builds a strict CCS.
- Ex, with $g_{Ex}(e[x]_p, (\emptyset, \emptyset)) = \{e\sigma', e\sigma''\}$ that generates a cover set. σ' (resp. σ'') is the substitution that replaces x by 0 (resp. $S(x')$ and x' is a fresh variable).

DELETE IDENTITY
$(E \cup \{t = t\},\ H) \vdash^{\mathsf{P}} (E \cup \Phi,\ H)$ if $\Phi = g_D(t = t, (\emptyset, \emptyset))$
REWRITE
$(E \cup \{e\},\ H) \vdash^{\mathsf{P}} (E \cup \Phi,\ H \cup \{e\})$ if $\Phi = g_R(e, (H, E))$
EXPAND
$(E \cup \{e\},\ H) \vdash^{\mathsf{P}} (E \cup \Phi, H)$ if $\Phi = g_{Ex}(e, (\emptyset, \emptyset))$

Figure 8.3: The inference system P.

Inference Rules Our inference system P, similar to the inference system I^b_s from Subsection 6.2.1, will contain the inference rules from Figure 8.3. For example, given the set of orientable axioms that define the addition over the naturals, $0 + x \to x$ and $S(x) + y \to S(x + y)$, P can prove $x + 0 = x$:
$(\{x + 0 = x\}, \emptyset) \vdash^{\mathsf{P}}_{Ex} (\{\underline{0 + 0} = 0, \underline{S(x') + 0} = S(x')\}, \emptyset) \vdash^{\mathsf{P}}_{R} (twice)$
$(\{0 = 0, S(x' + 0) = S(x')\}, \{0 + 0 = 0, S(x') + 0 = S(x')\}) \vdash^{\mathsf{P}}_{D}$
$(\{S(x' + 0) = S(x')\}, \{0 + 0 = 0, S(x') + 0 = S(x')\}) \vdash^{\mathsf{P}}_{Ex}$
$(\{S(\underline{0 + 0}) = S(0), S(S(x'') + 0) = S(S(x''))\}, \{0 + 0 = 0, S(x') + 0 = S(x')\}) \vdash^{\mathsf{P}}_{R}$
$(\{S(0) = S(0), S(\underline{S(x'') + 0}) = S(S(x''))\}, \{\ldots, S(x') + 0 = S(x')\}) \vdash^{\mathsf{P}}_{R}$
$(\{S(0) = S(0), S(\overline{S(x'')) = S(S(x''))}\}, \{\ldots\}) \vdash^{\mathsf{P}}_{D} (twice)(\emptyset, \{\ldots\})$,

where by $\vdash^{\mathsf{P}}_{D}$ (resp. $\vdash^{\mathsf{P}}_{R}$ and $\vdash^{\mathsf{P}}_{Ex}$) is denoted the application of DELETE IDENTITY (resp. REWRITE and EXPAND). The rewrite operations transform the underlined subterms with rewrite rules from the axioms, excepting the last operation that uses the instance $S(x'') + 0 \to S(x'')$ from the premises.
Instantiation Result P is and instance of A^1, as shown in Table 8.2.

P-rule	A^1-rule	RM, context, kind of CCS
DELETE	1-SIMPLIFY	D, $(\emptyset, \emptyset)$, empty
REWRITE	1-ADDPREMISE	I, (H, E), strict
EXPAND	1-SIMPLIFY	Ex, $(\emptyset, \emptyset)$, cover set

Table 8.2: P-rules as instances of A^1-rules.

Each column presents respectively for each P-rule the instantiated A^1-rule, together with the name of the reasoning module, the context and kind of the generated CCS. It is easy to observe from Table 8.1 and Figure 8.1 that the generated CCSs are compatible with and the contexts are included in the contexts of the corresponding A^1-rules. Therefore, P is sound and $Ax \models x + 0 = x$ can be concluded.

Simpler proofs of $x+0 = x$ can be obtained from instances of two-step A^2-rules, like EXPANDREWRITE in Figure 8.4. As an instance of 2-ADDPREMISE, it allows the addition of $x+0 = x$ to the set of premises directly from the first step of the proof for a later use as induction hypothesis in the subsequent REWRITE operations.

$$(E \cup \{e\},\ H) \vdash^{\mathsf{P}} (E \cup \Phi, H \cup \{e\})$$
$$\text{if } \Psi = g_{Ex}(e, (\emptyset, \emptyset)) \text{ and } \Phi = \bigcup_{e' \in \Psi} g_R(e', (H, E))$$

Figure 8.4: The ExpandRewrite rule.

8.2 Analysing and extending saturation-based inference systems

The proofs by saturation have an important place in the automated reasoning domain, for instance the Handbook of Automated Reasoning contains at least three chapters devoted to this topic [Bachmair and Ganzinger, 2001, Comon, 2001, Nieuwenhuis and Rubio, 2001]. They provide the means to detect the minimal counterexamples in the last state of any finite derivation. It is required that i) the formulas from the last state be *saturated*, i.e. no new information can be produced by the exhaustive application of the inference rules, and ii) the inference system be *refutationally complete*, i.e. the minimal counterexamples from any saturated formulas are 'easily detectable'. Such a minimal counterexample is usually the formula that is false in any model, denoted here by $\Box$, which is generated if and only if the saturated formulas are unsatisfiable.

To show the flexibility of our approach, we will prove the refutational completeness of a variant of A^1, denoted by A', for which the CCSs are universal. Indeed, the set of axioms is empty when applying the inference rules, but the argumentation for its refutational completeness is based on the existence of a *candidate model* [Bachmair and Ganzinger, 2001] (or generated model [Nieuwenhuis and Rubio, 2001]) for any saturated set of formulas, which plays the role of the axioms when defining counterexamples. We assume that it satisfies the following property:

Property 1 *Let $(E^0,\ \emptyset) \vdash^{\mathsf{A}'} \cdots \vdash^{\mathsf{A}'} (E^n,\ H^n)$ be a saturated A'-derivation such that I_{E^n} be a model for E^n. Then, an A'-inference rule is applicable on any formula from E^n containing a minimal counterexample other than $\Box$.*

Theorem 22 *(A'-refutational completeness) Let $(E^0,\ \emptyset) \vdash^{\mathsf{A}'} \cdots \vdash^{\mathsf{A}'} (E^n,\ H^n)$ be a saturated A'-derivation with $\Box \notin E^n$ and $I_{E^n} \models E^n$. If A' satisfies Property 1, for any ground formula ϕ from E^n, $I_{E^n} \models \phi$.*

Proof By contradiction, we assume that there is a counterexample in E^n. Theorem 20 still holds for A' as it is an instance of A^1. Therefore, there is a minimal counterexample γ in the derivation, other than $\Box$, that should be in E^n. On the other hand, Property 1 guarantees the applicability of an A'-rule on the formula containing γ, which contradicts the fact that the derivation is saturated. $\Box$

In the next subsection, the methodology will be used to analyse a paramodulation-based inference system.

8.2.1 Case study 1: a paramodulation-based inference system

The saturation-based inference system to analyse, denoted by G in Figure 8.5 and by $\mathcal{G}$ in [Nieuwenhuis and Rubio, 2001], reasons on ground Horn equational clauses of the form $e_1 \wedge \ldots \wedge e_n \Rightarrow$ or $e_1 \wedge \ldots \wedge e_n \Rightarrow e$, where $e, e_1, \ldots, e_n$ are equalities and $s \succ \tau(\Gamma)$ (resp. $s \succeq \tau(\Gamma)$) means that $s \succ u$ (resp. $s \succeq u$), for any term u from the equalities occurring in Γ. Based on ordered paramodulation, the rule Superposition Left (resp. Superposition Right) transforms (resp. strictly) maximal literals in the processed conjecture, while Equality Resolution eliminates from the negative part of a clause the identities involving its maximal term. It can be noticed that all these rules preserve the processed conjecture in the next derivation. The orderings over terms, $\prec$, and clauses, $\prec_c$, are Noetherian and total. The stability of $\prec_c$ is guaranteed because the formulas are ground. It has been shown in [Nieuwenhuis and Rubio, 2001] that G is refutationally complete and that Property 1 is satisfied. Moreover, it is (trivially) sound because all the processed conjectures, and therefore the minimal counterexamples, are preserved in the derivation.

SUPERPOSITION RIGHT $E \vdash^{\mathsf{G}} E \cup \{\Gamma', \Gamma \Rightarrow s[r]_p = t\}$
if $\Gamma' \Rightarrow l = r \in E$ s.t. $l \succ r$ and $l \succ \tau(\Gamma')$
and $\Gamma \Rightarrow s[l]_p = t \in E$ s.t. $s \succ t$ and $s \succ \tau(\Gamma)$

SUPERPOSITION LEFT $E \vdash^{\mathsf{G}} E \cup \{\Gamma', \Gamma, s[r]_p = t \Rightarrow \Delta\}$
if $\Gamma' \Rightarrow l = r \in E$ s.t. $l \succ r$ and $l \succ \tau(\Gamma')$
and $\Gamma, s[l]_p = t \Rightarrow \Delta \in E$ s.t. $s \succ t$ and $s \succeq \tau(\Gamma \wedge \Delta)$

EQUALITY RESOLUTION $E \vdash^{\mathsf{G}} E \cup \{\Gamma \Rightarrow \Delta\}$
if $\Gamma, s = s \Rightarrow \Delta \in E$ s.t. $s \succ \tau(\Gamma \wedge \Delta)$

Figure 8.5: The inference system G.

G-Variant

G is not adapted for the automated reasoning on long derivations. The time required to apply any of the superposition rules is directly proportional to the size of the current set of conjectures E. Or, this size increases with every inference step and therefore reduces the prover's performances. We propose a G-variant, denoted by G', that reduces considerably the size of E at the expense of less powerful rules by applying our methodology using universal CCSs.

A'-representation Firstly, an empty set of premises is appended to any G' state and the processed conjecture is no longer added to the new set of conjectures.

Reasoning Modules The new set of conjectures generated by any G'-rule is a strict CCS of the processed conjecture built by one of the following reasoning modules:

1. SR, used by the SUPERPOSITION RIGHT-variant, has the generation function, g_{SR}, defined as $g_{SR}(\Gamma \Rightarrow s[l]_p = t, (\{\Gamma' \Rightarrow l = r\}, \emptyset)) = \{\Gamma', \Gamma \Rightarrow s[r]_p = t\}$ since $\Gamma \Rightarrow s[l]_p = t \quad \succeq_c \quad \Gamma' \Rightarrow l = r$. Moreover, it builds a strict CCS because $\Gamma \Rightarrow s[l]_p = t \quad \succ_c \quad \Gamma', \Gamma \Rightarrow s[r]_p = t$, according to the definition of $\succ_c$ from [Nieuwenhuis and Rubio, 2001].

2. SL intervenes in the definition of the SUPERPOSITION LEFT-variant.
$g_{LR}((\Gamma, s[l]_p = t \Rightarrow \Delta), (\{\Gamma' \Rightarrow l = r\}, \emptyset)) = \{\Gamma', \Gamma, s[r]_p = t \Rightarrow \Delta\}$ is a strict CCS for similar reasons as above.

3. ER is used by EQUALITY REASONING-variant and has $g_{ER}((\Gamma, s = s \Rightarrow \Delta), (\emptyset, \emptyset)) = \{\Gamma \Rightarrow \Delta\}$ that generates a strict CCS because s is maximal and $\Gamma, s = s \Rightarrow \Delta \quad \succ_c \quad \Gamma \Rightarrow \Delta$.

Instantiation Result Any G'-rule is an instance of the 1-SIMPLIFY rule of A', as shown in Table 8.3. Its

G'-rule	A'-rule	RM, context, kind of CCS
SUPERPOSITION RIGHT	1-SIMPLIFY	SR, $(E, \emptyset)$, strict
SUPERPOSITION LEFT	1-SIMPLIFY	SL, $(E, \emptyset)$, strict
EQUALITY REASONING	1-SIMPLIFY	ER, $(\emptyset, \emptyset)$, strict

Table 8.3: G'-rules as instances of A'-rules.

columns have the same meaning as for Table 8.2.

G'-Variants In our setting, the ordering $\succ_c$ is not required to be total. Such non-total (partial) orderings can be built, for example, by generalising $<_e$ from equalities to conditional clauses. Representing $s_1 = t_1 \wedge \ldots \wedge s_n = t_n \Rightarrow l_1 = r_1 \vee \ldots \vee l_m = r_m$ as the multiset $\{s_1, t_1, \ldots, s_n, t_n, l_1, r_1, \ldots, l_m, r_m\}$ allows for the generation of smaller or equal clauses if: i) non-maximal terms of the processed conjecture are

reduced by Superposition Left and Superposition Right, and ii) identities of the form $s = s$ are eliminated by Equality Reasoning even if s is not maximal. These weakened versions of the G'-rules are still instances of 1-Simplify because it does not require strict CCSs.

Our methodology allows for a fine tuning for powerful and automated inference systems by considering two new rules for each G'-rule, with contexts defined by the corresponding one-step A'-inference rules:

1. As instances of 1-AddPremise, with respect to their original specifications, Superposition Left and Superposition Right are constrained to forbid the reduction of the maximal terms at the root position in order to build a strict CCS. There is no restriction for Equality Reasoning.
2. As instances of 1-Simplify, the rules are not constrained to reduce maximal terms.

An inference system containing only 1-AddPremise instances is as powerful as G: the processed conjecture is not saved in the new state as a conjecture, but as a premise. Notice that the G' and its variants satisfy Property 1 for similar reasons as G.

Theorem 23 *The inference system G' and its variants are sound and refutationally complete.*

Proof By Theorem 22 and the instantiation results. □

The saturation process always finishes on ground Horn clauses with any of G, G' and G'-variants. This is not the case for (Horn) clauses with variables, where the G'-variants are expected to be more effective. Techniques for lifting inference systems from the ground to the non-ground case are presented in [Nieuwenhuis and Rubio, 2001, Bachmair and Ganzinger, 2001].

8.2.2 Case study 2: a resolution-based inference system

This time, we will apply the methodology by extending a variant of A^1, denoted by A_s, with a new saturation-based rule.

$$\begin{array}{l}
\text{1-AddPremise } (E \cup \{\phi\},\ H) \ \vdash^{\mathsf{A}_s} \ (E \cup \Phi,\ H \cup \{\phi\}) \\
\quad \text{if } \{\phi\} \ \sqsubset_{(H,\ E)}\ \Phi \\
\text{1-Simplify } (E \cup \{\phi\},\ H) \ \vdash^{\mathsf{A}_s} \ (E \cup \Phi,\ H) \\
\quad \text{if } \{\phi\} \ \sqsubseteq_{(E \cup H,\ \emptyset)}\ \Phi \\
\text{1-Saturate } (E \cup \{\phi\},\ H) \ \vdash^{\mathsf{A}_s} \ (E \cup \Phi,\ H \cup \{\phi\}) \\
\quad \text{where } \Phi \text{ is } O^{>}(H_S, \phi)
\end{array}$$

Figure 8.6: The one-step A_s-inference rules.

The A_s inference system. The saturation-based inference system from Figure 8.6, denoted by A_s, consists of three rules. The first two (CCS-based) rules are particular cases of the rules from A^1, for which the CCSs are universal. The last rule, 1-Saturate, performs saturation-based reasoning before adding the processed conjecture to the premises. If Ψ is a set of premises, Ψ_S denotes all formulas from Ψ previously added by 1-Saturate. 1-Saturate firstly replaces ϕ by the new conjectures obtained by the *exhaustive* application of saturation-based operations $O^{>}$ (like those based on ordered resolution or paramodulation) between ϕ and formulas from H_S, then adds ϕ to H.

In order to prove the refutational completeness of A_s, the following admissible conditions should be satisfied: i) $\leq$ is a well-founded quasi-ordering over conjectures that is stable under substitutions and ii) for any saturated set of formulas, there exists a candidate model such that A_s satisfies the following property:

Property 2 *(A_s-**reducibility property for minimal counterexamples**) Let $(E^0,\ \emptyset) \vdash^{\mathsf{A}_s} \cdots \vdash^{\mathsf{A}_s} (\emptyset,\ H^n)$ be a saturated A_s-derivation such that $\square \notin H^n$ and $I_{H^n_S}$ be a candidate model for the set H^n_S. Then, for any premise ϕ having a minimal counterexample from H^n_S, there exists an inference rule in $O^{>}$ between ϕ and other formulas from H^n_S that produces a smaller counterexample.*

A_s is sound if the minimal counterexamples from $\cup_i E^i$ of any saturated A_s-derivation $(E^0,\ \emptyset) \vdash^{\mathsf{A}_s} \cdots \vdash^{\mathsf{A}_s} (\emptyset,\ H^n)$ are preserved in H^n_S. The next theorem states that A_s is sound and refutationally complete.

Theorem 24 *(**soundness and refutational completeness of** A_s) Let $(E^0,\ \emptyset) \vdash^{\mathsf{A}_s} \cdots \vdash^{\mathsf{A}_s} (\emptyset,\ H^n)$ be a saturated A_s-derivation such that $\square \notin H^n$ and $I_{H^n_S}$ be a candidate model for H^n_S. Then, A_s preserves in H^n_S any minimal counterexample from $\cup_i E^i$. Moreover, if A_s satisfies Property 2 then $I_{H^n_S} \models \phi$, for any ground instance ϕ of formulas from E^0.*

Proof To prove the soundness of A_s, we firstly show that H^n collects all minimal counterexamples from a derivation. Assume that γ is a minimal counterexample for $\cup_i E^i$ in $I_{H^n_S}$ of a saturated derivation $(E^0,\ \emptyset) \vdash^{\mathsf{A}_s} \cdots \vdash^{\mathsf{A}_s} (\emptyset,\ H^n)$ and let E^j (with $j \in [1..n-1]$) be the *last* set of conjectures in the derivation having a formula ϕ whose ground instance is a counterexample $\phi\tau$ equivalent to γ (so minimal). Such a set exists since the derivation ends with an empty set of conjectures. We will show that H^{j+1} has a $\phi\tau$-equivalent counterexample: i) if 1-AddPremise or 1-Saturate is applied on ϕ, then $\phi \in H^{j+1}$; ii) if the 1-Simplify rule $(E \cup \{\phi\}, H) \vdash^{\mathsf{A}_s} (E \cup \Phi), H)$ is applied on ϕ and $E^j = E \cup \{\phi\}$, then $\bigwedge((E \cup \Phi \cup H)_{\leq \phi\tau}) \Rightarrow \phi\tau$ since $\{\phi\} \sqsubseteq_{(E \cup H,\ \emptyset)} \Phi$. As $(E \cup \Phi \cup H)_{<\phi\tau}$ is true in $I_{H^n_S}$, the set $(E \cup \Phi \cup H)_{\sim\phi\tau}$ has a $\phi\tau$-equivalent counterexample. It cannot be a formula from $E \cup \Phi$ (which builds E^{j+1}), otherwise E^j is no longer such a last set. So, H (hence H^{j+1} and, finally, H^n) has a $\phi\tau$-equivalent counterexample.

The next step in the soundness proof of A_s is to show that for any minimal counterexample in H^n there exists a premise in H^n_S having an equivalent counterexample. Let $\phi \in H^n$ be a premise having a minimal counterexample $\phi\tau$. Notice that ϕ can be added to H^n either by 1-Saturate or 1-AddPremise. If 1-Saturate was applied, we are done. Otherwise, assume that 1-AddPremise was applied on ϕ, as in the transition $(E \cup \{\phi\},\ H) \vdash^{\mathsf{A}_s} (E \cup \Phi,\ H \cup \{\phi\})$. Since $\{\phi\} \sqsubset_{(H,\ E)} \Phi$, the relation $\bigwedge(H_{\leq\phi\tau} \cup E_{<\phi\tau} \cup \Phi_{<\phi\tau}) \Rightarrow \phi\tau$ holds in $I_{H^n_S}$. Therefore, $I_{H^n_S} \not\models H_{\leq\phi\tau} \cup E_{<\phi\tau} \cup \Phi_{<\phi\tau}$. On the other hand, $(H \cup E \cup \Phi)_{<\phi\tau}$ has no counterexample since $\phi\tau$ is minimal and $<$ is stable. Therefore, $I_{H^n_S} \not\models H_{\sim\phi\tau}$. By the stability of $\sim$, H should have a premise ψ that has a $\phi\tau$-equivalent counterexample, which has become premise before ϕ. The same reasoning can be applied to ψ as for ϕ, and it can be repeated until the derivation is scanned back to its beginning and eventually H becomes empty if 1-Saturate has not yet been applied. In this case, the only rule that can be applied is 1-Saturate and the processed conjecture is preserved in H^n_S.

The refutational completeness property is shown by contradiction. It is assumed that there exists a counterexample in E^0 and, therefore, a minimal counterexample γ $(\neq \square)$ in H^n_S, by the soundness property. On the other hand, Property 2 guarantees the existence of an inference rule from $O^>$ involving the set Ψ consisting from the premise from H^n_S having γ and other premises from H^n_S, that produces a counterexample smaller than γ. Due to the exhaustive application of saturation-based operations in any Inference Computation rule, such an inference should also occur in the derivation when applying 1-Saturate on the last added premise from Ψ. Contradiction. $\square$

It may be noticed from the proof of Theorem 24 that the soundness of A_s is not affected if i) the processed conjecture is preserved in the new set of conjectures, or ii) the following two inference rules are added:

- EliminatePremise
  $(E,\ H \cup \{\phi\}) \vdash^{\mathsf{A}_s} (E,\ H)$
- BackwardPremise
  $(E,\ H \cup \{\phi\}) \vdash^{\mathsf{A}_s} (E \cup \{\phi\},\ H)$,

where ϕ (in both rules) does not have minimal counterexamples.

Corollary 2 A_s *extended with the* EliminatePremise *and* BackwardPremise *rules is sound and refutationally complete when Property 2 is satisfied.*

In the following, we analyse the RP 'ordered resolution'-based inference system for non-ground clauses from [Bachmair and Ganzinger, 2001]. Before its presentation, we give some basic notions.

Basic notions. We recall that a *literal* is either an atomic formula A or its negation $\neg A$. A *clause* is a disjunction of literals $L_1 \vee \ldots \vee L_n$ that can be represented as the multiset $\{L_1, \ldots, L_n\}$. To show that a clause C contains a literal L, C is represented as $L \vee C'$, where C' is the *subclause* containing the rest of the literals from C. If L is a literal, then $\overline{L}$ is its *negation*. A clause C *subsumes* a clause D iff there exists a substitution σ such that $C\sigma \subseteq D$. If D does not subsume C, then C *properly* (or strictly) subsumes D.

We say that a formula ϕ is *redundant* with Φ if ϕ does not have any minimal counterexample in the candidate model I_Φ; for example, the standard redundancy criterion from [Bachmair and Ganzinger, 2001] defines a clause C as being redundant w.r.t. a set N of clauses if there exist some clauses $C_1, \ldots, C_k$ in N such that $C_1, \ldots C_k \models C$ and $C > C_i$, for all i with $1 \leq i \leq k$.

The RP system. RP, reproduced in Figure 8.7, is an abstract system used in a framework that models important classes of reasoning techniques like deduction, deletion and simplification. Its instances help to describe many proof strategies, as those implemented in the Otter system [McCune, 1994].

Each of its inference rules is a transition between triplets $(\mathcal{N} \mid \mathcal{P} \mid \mathcal{O})$ that store the "newly derived", the "processed" and the "old" clauses, respectively. The first three rules eliminate tautologies and subsumed clauses, while the next two rules simplify clauses by resolution. The sixth rule transfers clauses from $\mathcal{N}$ to $\mathcal{P}$. It suggests that $\mathcal{N}$ and $\mathcal{P}$ share the same set of clauses, but this partition is done for efficiency reasons; indeed, the newly derived clauses that are obtained by the application of the last rule are "processed" before migrating to $\mathcal{P}$. The last rule, Inference Computation, is *abstract*; $O_S^{\succ}(\mathcal{O}, C)$ performs all possible resolution-based operations between C and clauses from $\mathcal{O}$. The clauses from $\mathcal{O}$ are chosen according to the selection function S, given as parameter, that helps to define a wide variety of proof strategies.

A_s-representation. Our analysis is based on the aforementioned suggestion. The new rules are obtained by replacing in the RP-rules i) the sets $\mathcal{N}$ and $\mathcal{P}$ by the set of conjectures E, and ii) $\mathcal{O}$ by the premises H. The resulted inference system, denoted by RP′, is displayed in Figure 8.8. Clause Processing is now useless and Backward Reduction for $\mathcal{P}$ becomes redundant with Forward Reduction, so they will not be mentioned in RP′.

Admissible Conditions. The inference system RP′, as RP, is abstract. In [Bachmair and Ganzinger, 2001], RP has many refutationally complete instances, for example when $O_S^{\succ}(H, C)$ applies the rules from the *General Ordered Resolution* inference system (see Theorem 5.5 in [Bachmair and Ganzinger, 2001]). The Property 2 is also satisfied, according to Theorem 5.4 from [Bachmair and Ganzinger, 2001]. The quasi-ordering $\preceq$ over clauses is stable under substitutions and well-founded, having its strict part $\prec$ total on ground clauses.

Reasoning Modules. We define the following reasoning modules:

1. TD, based on *Tautology Deletion*, can build an empty CCS. Its generation function, g_{TD}, is defined as $g_{TD}(C, (\emptyset, \emptyset)) = \emptyset$ if C is a tautology.
2. S, based on *Subsumption*, can build an empty CCS: $g_S(C, (\mathsf{C}^1, \emptyset)) = \emptyset$ if there exists a clause in C^1 that subsumes C.
3. SS, based on *Strict Subsumption*, can build an empty CCS, too: $g_{SS}(C, (\emptyset, \mathsf{C}^2)) = \emptyset$ if there exists a clause in C^2 that properly subsumes C.
4. SR, based on *Subsumption Resolution*, can build a strict CCS: $g_{SR}(C \vee L, (\mathsf{C}^1, \emptyset)) = \{C\}$ if there exists $D \vee L'$ in C^1 such that $\overline{L} = L'\sigma$ and $D\sigma \subseteq C$.

Instantiation test. Table 8.4 displays the non-trivial RP′-rules that instantiate 1-Simplify from Figure 8.6, together with the corresponding reasoning modules, context content and kind of their built CCSs. Any reasoning module provides a CCS compatible with the general CCS of 1-Simplify, and context smaller than or equal to $(E \cup H, \emptyset)$, i.e. the context value defined in 1-Simplify. Also notice

1. Tautology Deletion:

$\mathcal{N} \cup \{C\} \mid \mathcal{P} \mid \mathcal{O} \vdash^{\mathsf{RP}} \mathcal{N} \mid \mathcal{P} \mid \mathcal{O}$

if C is a tautology

2. Forward Subsumption:

$\mathcal{N} \cup \{C\} \mid \mathcal{P} \mid \mathcal{O} \vdash^{\mathsf{RP}} \mathcal{N} \mid \mathcal{P} \mid \mathcal{O}$

if some clause in $\mathcal{P} \cup \mathcal{O}$ subsumes C

3. Backward Subsumption:

- for $\mathcal{P}$: $\mathcal{N} \mid \mathcal{P} \cup \{C\} \mid \mathcal{O} \vdash^{\mathsf{RP}} \mathcal{N} \mid \mathcal{P} \mid \mathcal{O}$
- for $\mathcal{O}$: $\mathcal{N} \mid \mathcal{P} \mid \mathcal{O} \cup \{C\} \vdash^{\mathsf{RP}} \mathcal{N} \mid \mathcal{P} \mid \mathcal{O}$

if some clause in $\mathcal{N}$ properly subsumes C

4. Forward Reduction:

$\mathcal{N} \cup \{C \vee L\} \mid \mathcal{P} \mid \mathcal{O} \vdash^{\mathsf{RP}} \mathcal{N} \cup \{C\} \mid \mathcal{P} \mid \mathcal{O}$

if there is $D \vee L'$ in $\mathcal{P} \cup \mathcal{O}$ s. t. $\overline{L} = L'\sigma$ **and** $D\sigma \subseteq C$

5. Backward Reduction:

- for $\mathcal{P}$: $\mathcal{N} \mid \mathcal{P} \cup \{C \vee L\} \mid \mathcal{O} \vdash^{\mathsf{RP}} \mathcal{N} \cup \{C\} \mid \mathcal{P} \mid \mathcal{O}$
- for $\mathcal{O}$: $\mathcal{N} \mid \mathcal{P} \mid \mathcal{O} \cup \{C \vee L\} \vdash^{\mathsf{RP}} \mathcal{N} \cup \{C\} \mid \mathcal{P} \mid \mathcal{O}$

if there is $D \vee L'$ in $\mathcal{N}$ s. t. $\overline{L} = L'\sigma$ **and** $D\sigma \subseteq C$

6. Clause Processing:

$\mathcal{N} \cup \{C\} \mid \mathcal{P} \mid \mathcal{O} \vdash^{\mathsf{RP}} \mathcal{N} \mid \mathcal{P} \cup \{C\} \mid \mathcal{O}$

7. Inference Computation:

$\emptyset \mid \mathcal{P} \cup \{C\} \mid \mathcal{O} \vdash^{\mathsf{RP}} \mathcal{N} \mid \mathcal{P} \mid \mathcal{O} \cup \{C\}$

where $\mathcal{N}$ collects the clauses generated by $O^{\succ}_{S}(\mathcal{O}, C)$

Figure 8.7: RP: the original inference system with triplets as proof states.

that Inference Computation can instantiate 1-Saturate. The rule Backward Subsumption for H is an instance of EliminatePremise; whenever a clause C is strictly subsumed by a clause C' and C has a counterexample, then C' has a smaller one (so the premises to which EliminatePremise is applied never have minimal counterexamples). The rule Backward Reduction is the result of the application of BackwardPremise, followed by Forward Reduction.

This instantiation result, the satisfaction of the admissible conditions and Corollary 2 conclude that RP′ (and therefore RP) is sound and refutationally complete.

RP- and RP′-variants. The instantiation result and Theorem 21 suggest that, theoretically, the contexts from Tautology Deletion and Backward Subsumption for E can be soundly extended to $(E \cup H, \emptyset)$. However, from a practical point of view, this is useless since i) a tautology remains a tautology in any context, and ii) the extended Backward Subsumption would become identical to Forward Subsumption. In fact, Backward Subsumption is kept in RP for efficiency reasons.

An interesting extension would be to add instances of 1-AddPremise to RP′. For example, when

Tautology Deletion (T.D.):

$(E \cup \{C\}, H) \vdash^{\mathsf{RP}'} (E, H)$

if C is a tautology

Forward Subsumption (F.S.):

$(E \cup \{C\}, H) \vdash^{\mathsf{RP}'} (E, H)$

if some clause in $E \cup H$ subsumes C

Backward Subsumption (B.S.):

- for E: $(E \cup \{C\}, H) \vdash^{\mathsf{RP}'} (E, H)$

- for H: $(E, H \cup \{C\}) \vdash^{\mathsf{RP}'} (E, H)$

if some clause in E properly subsumes C

Forward Reduction (F.R.):

$(E \cup \{C \vee L\}, H) \vdash^{\mathsf{RP}'} (E \cup \{C\}, H)$

if there is $D \vee L'$ in $E \cup H$ s. t. $\overline{L} = L'\sigma$ **and** $D\sigma \subseteq C$

Backward Reduction (B.R.):

$(E, H \cup \{C \vee L\}) \vdash^{\mathsf{RP}'} (E \cup \{C\}, H)$

if there is $D \vee L'$ in E s. t. $\overline{L} = L'\sigma$ **and** $D\sigma \subseteq C$

Inference Computation (I.C.):

$(E \cup \{C\}, H) \vdash^{\mathsf{RP}'} (E \cup \Phi, H \cup \{C\})$
where Φ collects the clauses generated by $O_S^{\succ}(H, C)$

Figure 8.8: RP': the inference system RP with proof states under the form *(conjectures, premises)*.

Forward Reduction builds a strict CCS in the context (H, E), it can send the processed clause directly to H, without performing Inference Computation. Notice that the original RP' may obtain the same result by firstly applying Inference Computation followed by Backward Reduction, which is less efficient due to the exhaustive application of $O_S^{\succ}$ rules and to the addition of the resulted formulas to the set of conjectures. Translated back to RP, such a rule can be $\mathcal{N} \cup \{C \vee L\} \mid \mathcal{P} \mid \mathcal{O} \vdash^{\mathsf{RP}} \mathcal{N} \cup \{C\} \mid \mathcal{P} \mid \mathcal{O} \cup \{C \vee L\}$ if there is $D \vee L'$ in $\mathcal{O}$ such that $\overline{L} = L'\sigma$, $D\sigma \subseteq C$ and L is maximal w.r.t. any literal in C. In this case, the first argument of $O_S^{\succ}$ in Inference Computation should be limited only to the formulas from $\mathcal{O}$ to which Inference Computation was previously applied.

8.3 Conclusions

The presented work has both applicative and theoretical interests. We have proposed a framework and a methodology to design and analyse inference systems implementing the formula-based Noetherian induction principle. A stronger connection between two important fields of the automated deduction, the implicit induction and saturation-based theorem proving, is established. Their logic is captured by an abstract inference system: to witness, the inference systems from the given examples have been shown to be instantiations of it. The orderings over formulas need only to be Noetherian and stable under substitutions to satisfy the soundness property, as can be observed from the proof of Theorem 20. Thanks to the clear separation between the logic and computation provided by the framework, it can be

RP′-rule	RM	Context Content	Kind of CCS
T.D.	TD	$(\emptyset, \emptyset)$	empty
F.S.	S	$(E \cup H, \emptyset)$	empty
B.S.(FOR E)	SS	$(\emptyset, E)$	empty
F. R.	SR	$(E \cup H, \emptyset)$	strict

Table 8.4: RP′-rules instantiating the 1-SIMPLIFY-rule.

concluded that other ordering properties encountered in the literature, as the totality, are required either in the computation part or to satisfy additional properties like Property 1.

The abstract systems A^1 and A^2 are landmarks for the formula-based Noetherian induction inference systems. As shown for the system A from Section 6.4 [Stratulat, 2000a], they provide the maximal set of induction hypotheses w.r.t. other similar implicit induction abstract systems. We expect similar results for the saturation-based inference systems. Usually, the one- and two-step inference rules are sufficient to apply the methodology, but the framework is able to provide 'more than two'-step rules when necessary. The computation process has the advantage to be parameterisable; it is performed by reasoning modules that allow for the modular design of the concrete inference systems.

The methodology indicates how to instantiate the abstract system with specific reasoning techniques without affecting its soundness. The soundness proofs of concrete procedures become advantageously simpler because the soundness of the abstract systems was established once for all. They are mainly reduced to show that any rule is the instance of an abstract inference rule.

Future Works

Our work opens opportunities for further developments and new ideas. As example, we will develop in Section C.1 a new proposition for certifying cyclic reasoning by converting cyclic proofs to explicit induction proofs. It is based on the FOL_{ID} logical framework and is more general than that presented in Subsection 3.2.4. The document finishes by presenting other, less detailed, lines of research.

C.1 Detailed project: certification of cyclic reasoning by converting cyclic to explicit induction proofs

Another way to certify cyclic reasoning using Coq is to convert cyclic proofs to explicit induction proofs. In Subsection 3.2.4, we have presented a method for converting to explicit induction proofs cyclic formula-based Noetherian induction proofs that have only 1-cycles and the IHs are instances of the root formula. This approach is rather limited; for example, the cyclic Coq pre-proofs from Example 3 can not be processed because either some IHs are not instances of the root formula or some cycles are not 1-cycles.

In the following, we will present a new and more general conversion procedure, based on the FOL_{ID} logical framework. The reason for choosing FOL_{ID} is two-fold: i) it offers a crispy clear formal framework to explain the conversion procedure, and ii) the FOL_{ID} reasoning can be reproduced by Coq scripts.

C.1.1 The LKID and LKID_a explicit induction inference systems

The classical explicit induction inference system for FOL_{ID} is the LKID system [Brotherston and Simpson, 2011]. It includes the LK's rules from Figure 7.1, the $\text{CLKID}^{\omega}_{N}$'s unfold rule, as well as a new rule, denoted by (Ind), which represents a left introduction rule for an inductive atom $P_j(\overline{t}')$:

$$\frac{\text{minor premises} \qquad \Gamma, F_j(\overline{t}') \vdash \Delta}{\Gamma, P_j(\overline{t}') \vdash \Delta}\ (Ind\ P_j)$$

A *minor premise* is built from each axiom defining a predicate P that is P_j or mutually dependent with P_j. The minor premise corresponding to (7.7) is

$$\Gamma, Q_1(\overline{u}_1), \ldots, Q_h(\overline{u}_h), G_1(\overline{t}_1), \ldots, G_l(\overline{t}_l) \vdash F(\overline{t}'), \Delta \tag{C.2}$$

if (7.7) and $\Gamma, P_j(\overline{t}') \vdash \Delta$ do not share variables (otherwise, the variables from (7.7) can be renamed accordingly), and $G_1, \ldots, G_l$, F are *IH formulas* associated to $P_1, \ldots, P_l$, P, respectively. $\Gamma, F_j(\overline{t}') \vdash \Delta$ is called *major premise*.

LKID has been shown sound, i.e., every LKID inference rule r is sound (see Proposition 3.5 from [Brotherston and Simpson, 2011]) such that the conclusion of r holds whenever its premises hold.

For our purpose, we assume that there is a non-empty set of *admissible* inductive atoms.

Definition 13 (admissible inductive atom) *An inductive atom $P(\overline{t})$ is* admissible *if the sequent $\vdash P(\overline{t})$ is true.*

Admissible inductive atoms can be used to build the following new rule:

Definition 14 (the (Adm) rule) *(Adm) is the 0-premise rule for the right introduction of any admissible inductive atom $P(\overline{t})$:*

$$\frac{}{\Gamma \vdash P(\overline{t}), \Delta}\ (Adm\ P(\overline{t}))$$

Lemma 12 (soundness of (Adm)) *The (Adm) rule is sound.*

Proof By Definition 13. □

$$\dfrac{\dfrac{}{Nx \vdash_3 Qx0}\,(R.(C.5))}{\dfrac{Nx, y = 0 \vdash_2 Qxy}{Nx, \underline{Ny} \vdash_1 Qxy\ (\dagger)}\,(Gen)} \quad \dfrac{\dfrac{Nx \vdash_7 Px\ (*\mathbf{2})}{Nx, Nz \vdash_5 Qxsz}\,(Gen) \qquad \dfrac{Nx, Ny \vdash_9 Qxy\ (\dagger\mathbf{1})}{Nz, Nx \vdash_8 Qxz}\,(Subst)}{Nx, y = sz, Nz \vdash_4 Qxy}\,(R.(C.6)), (Wk)\,(Case\ N(y))$$

$$\dfrac{\dfrac{}{N0 \vdash_{13} P0}\,(R.(C.3))}{\dfrac{Nx, x = 0 \vdash_{12} Px}{Nx, \underline{Nx} \vdash_{11} Px}\,(Gen)} \quad \dfrac{\dfrac{Nx \vdash_{17} Px\ (*\mathbf{1})}{Nz \vdash_{16} Pz}\,(Subst) \qquad \dfrac{Nx, Ny \vdash_{19} Qxy\ (\dagger\mathbf{2})}{Nz, Nsz \vdash_{18} Qzsz}\,(Subst)}{Nsz, Nz \vdash_{15} Psz}\,(R.(C.4)), (Wk)\,(Gen)}{Nx, x = sz, Nz \vdash_{14} Px}\,(Case\ N(x))$$

Figure C.9: The input pre-proof tree-set.

Let P_a be an admissible inductive predicate, recursively defined by a set of axioms that can be either unconditional (of the form of $P_a(\bar{t})$) or conditional (of the form $\wedge_{m=1}^{l} P_a(\bar{t}_m) \Rightarrow P_a(\bar{t})$, for some natural $l > 0$). Based on the (Adm) rule, the derived rule (Ind_a)

$$\dfrac{F(\bar{t}_1), \ldots, F(\bar{t}_l) \vdash F(\bar{t}_i) \quad \ldots}{\vdash F(\bar{t})}\,(Ind_a\ P_a)$$

abbreviates

$$\dfrac{\dfrac{}{\vdash P_a(\bar{t})}\,(Adm\ P_a(\bar{t})) \quad \dfrac{F(\bar{t}_1), \ldots, F(\bar{t}_l) \vdash F(\bar{t}_i) \ \ldots \ \dfrac{}{F(\bar{t}) \vdash F(\bar{t})}\,(Ax)}{P_a(\bar{t}) \vdash F(\bar{t})}\,(Ind\ P_a)}{\vdash F(\bar{t})}\,(Cut), (Wk)$$

where $F(\overline{x})$ is the IH formula associated to P_a for the (Ind) step.

Definition 15 (LKID$_a$, LKID$_a$ proof) *LKID$_a$ is the particular instance of LKID i) for which the (Ind) rule is replaced by the (Ind_a) rule, and ii) extended with the (Adm) rule. A finite LKID$_a$-derivation tree is a* proof *if all branches end in a 0-premise rule.*

Theorem 25 (soundness of LKID$_a$) *Every sequent $\Gamma \vdash \Delta$ is true if there is an LKID$_a$ proof of $\Gamma \vdash \Delta$.*

Proof LKID$_a$ is a particular version of the sound LKID inference system, that was extended with the sound (Adm) rule. □

We will show how to convert CLKID$_N^\omega$ pre-proofs to LKID$_a$ proofs. The conversion procedure takes as input i) a CLKID$_N^\omega$ pre-proof tree-set in normal form, and ii) one of its root sequents S. It outputs a sequence of LKID$_a$ proofs ending with an LKID$_a$ proof of S.

C.1.2 Overview of the procedure

The input pre-proof tree-set ($\mathcal{MD}$, $\mathcal{MR}$) can be seen as a digraph $\mathcal{P}$, as in Subsection 7.2.2. As example, we will consider the pre-proof tree-set corresponding to the digraph from Example 4 concerning the P&Q example.

Example 29 *Let us introduce the inductive definitions of P and Q from the P&Q example:*

$$P(0) \quad \text{(C.3)} \qquad\qquad Q(x,0) \quad \text{(C.5)}$$
$$P(x) \wedge Q(x,s(x)) \Rightarrow P(s(x)) \quad \text{(C.4)} \qquad\qquad P(x) \wedge Q(x,y) \Rightarrow Q(x,s(y)) \quad \text{(C.6)}$$

A pre-proof tree-set, rooted by the sequents $Nx \vdash Px$ and $Nx, Ny \vdash Qxy$, is illustrated in Figure C.9. For convenience, the sequents are indexed. The node labelled by the sequent $\Gamma \vdash_i \Delta$ is denoted as N^i.

We denote by $S(N)$ the sequent labeling any node N from $\mathcal{MD}$. We define a partial order $\leq_{\mathcal{R}}$ over the set of *root* nodes of $\mathcal{MD}$, as follows: for any two distinct nodes N^1 and N^2, we have $N^1 <_{\mathcal{R}} N^2$ if N^1 and N^2 are not in the same SCC and N^1 can be reached from N^2. We have $N^1 =_{\mathcal{R}} N^2$ if both N^1 and N^2 are in the same SCC. The conversion procedure visits each root from $\mathcal{MD}$ increasingly, by using as precedence the partial order $\leq_{\mathcal{R}}$. Let N_r be some root. The explicit proof for $S(N_r)$ is recursively built, as follows:

1. let us assume that N_r is the only node of a singleton SCC. If N_r is a node labeled by a 0-premise rule, the output is the LKID$_a$ proof consisting of $S(N_r)$ to which the 0-premise rule is applied. If N_r is a bud, let N_c be its companion. Since N_c is different from N_r, it is $<_{\mathcal{R}}$-smaller than N_r. By induction hypothesis, its LKID$_a$ proof is already built at this stage. The output is the LKID$_a$ proof generated in each of the cases.
2. let us now assume that N_r is a root node from a non-singleton SCC π. Then,
   (a) define an explicit induction schema to be applied on the *conjunction sequent* representing the sequent having an empty antecedent part and the conjunction of the implication formulas of the root sequents from π as the only formula from the succedent part;
   (b) build the explicit induction proof of the conjunction sequent, starting by applying the induction schema from step (a), then proving each induction case (IC) by following proof steps from π. The sequents labelling the buds with $<_{\mathcal{R}}$-smaller companions can be proved using the proofs of the companions;
   (c) prove $S(N_r)$ from the proof of the conjunction sequent.

The conversion procedure outputs the LKID$_a$ proofs for each sequent labelling $<_{\mathcal{R}}$-greatest root nodes or for the conjunction sequent of their non-singleton SCC, if case 2 applies. Next, we detail the operations executed at case 2.

C.1.3 Generating the explicit induction schema

Let us assume that the SCC with cycles π has k (> 0) roots $N^{i_1}, \ldots, N^{i_k}$, where $i_1, \ldots, i_k \in [1..k]$ is a permutation of the values from $[1..k]$. $F_\pi(\overline{t})$ denotes the *conjunction formula* $\bigwedge_{j=1}^{k} \mathcal{C}(S(N^{i_j}))$ associated to π. Its conjuncts do not share free variables (otherwise, the free variables are conveniently renamed), where $\forall j \in [1..k]$, $S(N^{i_j}) \equiv \Gamma^{i_j}, P^1_{i_j}(\overline{t^1_{i_j}}), \ldots, P^{n_{i_j}}_{i_j}(\overline{t^{n_{i_j}}_{i_j}}) \vdash \Delta^{i_j}$ such that Γ^{i_j} has no inductive atoms, $\overline{t} \equiv (\overline{t^1_{i_1}}, \ldots, \overline{t^{n_{i_1}}_{i_1}}, \ldots, \overline{t^1_{i_k}}, \ldots, \overline{t^{n_{i_k}}_{i_k}})$, and $n_{i_1}, \ldots, n_{i_k} \in \mathbb{N}^*$.

The explicit induction step consists in the application of the $(Ind_a\ P_\pi)$ rule on the *conjunction sequent* $\vdash F_\pi(\overline{t})$, given that $P_\pi(\overline{t})$ is an admissible inductive atom and P_π is a new inductive predicate symbol associated to π. The axioms defining P_π are built from the analysis of the tree derivations rooted by $N^{i_1}, \ldots, N^{i_k}$. Each axiom corresponds to some (IC) of the induction schema built from the *composition* of *individual* induction schemas defined for each root sequent $S(N^{i_j})$ ($j \in [1..k]$). This step is detailed in the following.

Building the individual induction schemas

An *induction schema* for a root sequent S is a collection of *induction cases* (ICs) that attach a (potentially empty) set of sequents, called *induction hypotheses* (IH), to an instance of S, called *induction conclusion*. The instantiating substitutions are denoted as *cumulative* substitutions consisting of the composition of substitutions used in the (Gen)-steps encountered when traversing a given path, as explained in Subsection 7.2.2.

Example 30 *The cumulative substitutions for the paths leading a root to a terminal node in the pre-proof tree-set from Figure C.9 are:*

- $\{x \mapsto x; y \mapsto 0\}$ *for the path* $[N^1, N^2, N^3]$*;*
- $\{x \mapsto x; y \mapsto sz; z \mapsto z\}$ *for* $[N^1, N^4, N^5, N^7]$ *and* $[N^1, N^4, N^5, N^8, N^9]$*;*
- $\{x \mapsto 0\}$ *for the path* $[N^{10}, N^{11}, N^{12}, N^{13}]$*, and*
- *the substitution* $\{x \mapsto sz; z \mapsto z\}$ *for the paths* $[N^{10}, N^{11}, N^{14}, N^{15}, N^{16}, N^{17}]$ *and* $[N^{10}, N^{11}, N^{14}, N^{15}, N^{18}, N^{19}]$*.*

Definition 16 (individual induction schema) *Let N_r be some root from π. The* individual induction schema *for $S(N_r)$ is built from a set of ICs. A new IC with the induction conclusion $S(N_r)[\theta]$ is generated for each path p leading N_r to a terminal node N_t of the tree derivation rooted by N_r, where θ is the cumulative substitution for p. When N_t is a bud whose companion N_h is from π, the explicit induction hypothesis $S(N_h)[\delta]$ is attached to the induction conclusion $S(N_r)[\theta]$, where δ is either i) the substitution underlying some (Subst) rule having $S(N_t)$ as premise, if any, or ii) $\sigma_{id}^{S(N_h)}$ if $S(N_t)$ is the premise of a rule other than (Subst).*

As shown in Example 30, different paths may have attached the same cumulative substitution. Hence, an induction conclusion may have attached several explicit IHs.

Example 31 (cont. Example 30) *By using the previously computed cumulative substitutions, the ICs of the individual induction schema for*

- $Nx \vdash Px$ *are: i)* $N0 \vdash P0$*, and ii)* $Nsz \vdash Psz$ *with the attached explicit IHs* $Nz \vdash Pz$ *and* $Nz, Nsz \vdash Qzsz$*;*
- $Nx, Ny \vdash Qxy$ *are: i)* $Nx, N0 \vdash Qx0$*, and ii)* $Nx, Nsz \vdash Qxsz$ *with the attached explicit IHs* $Nx \vdash Px$ *and* $Nz, Nx \vdash Qxz$*.*

Generating the axioms for P_π

P_π is a single recursive inductive predicate defined by the smallest set of conditional axioms of the form

$$\textbf{premises} \Rightarrow P_\pi(\bar{t}_{i_1}, \ldots, \bar{t}_{i_k}) \tag{C.7}$$

and unconditional axioms of the form $P_\pi(\bar{t}_{i_1}, \ldots, \bar{t}_{i_k})$, where the size of $\bar{t}_{i_j}$ $(j \in [1..k])$ is the sum of the sizes of each $\overline{t_{i_j}^l}$ $(l \in [1..n_{i_j}])$. Each of the sequents $S(N^{i_j})[\theta]$ is the conclusion of an individual IC defined for $S(N^{i_j})$, where $\bar{t}_{i_j}$ is $(\overline{t_{i_j}^1}[\theta], \ldots, \overline{t_{i_j}^{n_{i_j}}}[\theta])$ $(j \in [1..k])$. For each of its attached explicit IHs, there is a premise of (C.7) defined as follows. If the explicit IH is the sequent $S(N^{i_r})[\delta]$ $(r \in [1..k])$, the premise is the induction conclusion of (C.7) for which the rth (sub)vector of its arguments is replaced by $(\overline{t_{i_r}^1}[\delta], \ldots, \overline{t_{i_r}^{n_{i_r}}}[\delta])$. The premises of an axiom having replaced distinct (sub)vectors may be factorized in one premise, which can lead to different definitions for P_π. For example, if two premises replace respectively the uth and vth subvector, with $u, v \in [1..k]$ and $u < v$, they can be factorized to give the premise $P_\pi(\bar{t}_{i_1}, \ldots, \bar{t}_{i_{u-1}}, \bar{s}_{i_u}, \bar{t}_{i_{u+1}}, \ldots, \bar{t}_{i_{v-1}}, \bar{s}_{i_v}, \bar{t}_{i_{v+1}}, \ldots, \bar{t}_{i_k})$.

Example 32 (cont. Example 31) *P_π can be defined by the axioms*

$$P_\pi(0, x, 0) \tag{C.8}$$

$$P_\pi(x, x, z') \Rightarrow P_\pi(0, x, s(z')) \tag{C.9}$$

$$P_\pi(z, z, s(z)) \Rightarrow P_\pi(s(z), x, 0) \tag{C.10}$$

$$P_\pi(z, z, s(z)) \wedge P_\pi(x, x, z') \Rightarrow P_\pi(s(z), x, s(z')) \tag{C.11}$$

issued from the individual induction schemas for $Nx \vdash Px$ and $Nx, Ny \vdash Qxy$, by factorising the premises. For example, the premise of the axiom (C.10) was factorised from the premises $P_\pi(z, x, 0)$ and $P_\pi(s(z), z, s(z))$. Another definition of P_π can be given with a different factorisation of the premises for (C.11) when (C.11) is replaced by $P_\pi(x, z, s(z)) \wedge P_\pi(z, x, z') \Rightarrow P_\pi(s(z), x, s(z'))$.

$$\dfrac{\dfrac{\dots\dots\dots\dots\dots\dots\dots\dots}{\mathcal{C}(\Gamma_j[\delta] \vdash \Delta_j[\delta]), \Gamma_j[\delta] \vdash \Delta_j[\delta]}}{\dots, \mathcal{C}(\Gamma_j[\delta] \vdash \Delta_j[\delta]), \dots, \Gamma_j[\delta] \vdash \Delta_j[\delta]}\,(Wk) \qquad \dfrac{\dfrac{\dfrac{\textit{proof as for } \Gamma \vdash \Delta}{\Gamma_j \vdash \Delta_j}}{\Gamma_j[\delta] \vdash \Delta_j[\delta]}\,(Subst)}{\Gamma', \Gamma_j[\delta] \vdash \Delta_j[\delta]}\,(Wk) \qquad \dfrac{}{\Gamma'' \vdash \Delta''}\,(0\text{-premise})$$

N_t: bud (companion $\in \pi$) $\quad$ N_t: bud (companion $\notin \pi$) $\quad$ N_t: leaf

(one of the three scenarios from above are executed, depending on the nature of N_t)

$$\vdots$$

(*applying successively the rules from the nodes of* p, according to Lemma 9)

$$\vdots$$

$$\dfrac{\mathcal{C}(S(N_l^1))[\delta^1], \dots, \mathcal{C}(S(N_l^p))[\delta^p], \Gamma[\theta] \vdash \Delta[\theta]}{\mathcal{C}(S(N_l^1))[\delta^1], \dots, \mathcal{C}(S(N_l^p))[\delta^p] \vdash \mathcal{C}(\Gamma \vdash \Delta)[\theta]}\,(\vee R),(\neg R)$$

(a) Proving an individual IC built by following the path p leading N_r to some terminal node N_t.

$$\dfrac{\{\bigcup_{i=1}^{p}\{\mathcal{C}(S(N_l^i))[\delta^i]\} \vdash \mathcal{C}(\Gamma \vdash \Delta)[\theta] \mid \text{`}(\Gamma \vdash \Delta)[\theta] \text{ attaches } \bigcup_{i=1}^{n}\{S(N_l^i)[\delta^i]\}\text{' is an indiv. IC}\}}{\vdash \mathcal{C}(S(N_{i_1})) \wedge \ldots \wedge \mathcal{C}(S(N_{i_k}))}$$

(b) The new subgoals after applying $(Ind_a\ P_\pi)$, $(\wedge L)$, $(\wedge R)$ and (Wk) on the conjunction sequent.

$$\dfrac{\dfrac{\dfrac{\dots\dots\dots\dots\dots}{\mathcal{C}(\Gamma \vdash \Delta), \Gamma \vdash \Delta}}{\bigwedge_{\text{root } N \text{ from } \pi} \mathcal{C}(S(N)), \Gamma \vdash \Delta}\,(\wedge L),(\text{Wk}) \qquad \dfrac{\dfrac{(\textit{cont. Figure C.10b})}{\vdash \mathcal{C}(S(N_{i_1})) \wedge \ldots \wedge \mathcal{C}(S(N_{i_k}))}\,(Wk)}{\Gamma \vdash \bigwedge_{\text{root } N \text{ from } \pi} \mathcal{C}(S(N)), \Delta}\,(\text{Wk})}{\Gamma \vdash \Delta}\,(\text{Cut})$$

(c) Building the proof of the conjunction sequent for the SCC with cycles π.

Figure C.10: The main steps of the LKID$_a$ proof of $S(N_r) \equiv \Gamma \vdash \Delta$.

C.1.4 Generating the explicit induction proof

The explicit induction proof of the conjunction sequent starts by a unique explicit induction step using $(Ind_a\ P_\pi)$. Each IC generates a new conjecture whose proof script can be directly issued by following the paths used to build the individual ICs producing the IC. This is mainly due to Lemma 9.

Applying the (Ind_a) rule on the conjunction sequent

The scenario for applying $(Ind_a\ P_\pi)$ on the conjunction sequent is given in Figure C.10b, under the assumption that P_π is admissible. The double line denotes a sequence of rule applications following $(Ind_a\ P_\pi)$ that transform each premise of $(Ind_a\ P_\pi)$ into the individual ICs that helped to built it. It firstly deletes the conjunction symbols from the antecedent part using successive applications of $(\wedge L)$ rules, then the conjunction symbols from the succedent part using the $(\wedge R)$ rule. The antecedent formulas from each new sequent, that are not among the explicit IHs attached to the induction conclusion from the succedent, are finally deleted by the (Wk) rule. The IC '$(\Gamma \vdash \Delta)[\theta]$ attaches $S(N_l^1)[\delta^1] \dots S(N_l^p)[\delta^p]$' can be formalised as the sequent $\mathcal{C}(S(N_l^1))[\delta^1], \dots, \mathcal{C}(S(N_l^p))[\delta^p] \vdash \mathcal{C}(\Gamma \vdash \Delta)[\theta]$.

Example 33 (cont. Example 32) *Let us assume that N is admissible. We can simplify the LKID$_a$ proof by considering the weaker versions of $\mathcal{C}(Nu \vdash Pu)$ (resp., $\mathcal{C}(Nx, Ny \vdash Qxy)$) as Pu, for any term u (resp., Qxy, for any terms x and y). The beginning part of the LKID$_a$ proof of the conjunction sequent $\vdash Pu \wedge Qxy$ is illustrated in Figure C.11, under the assumption that P_π is also admissible. The missing proof parts for the sequents denoted by [*] and [**] are given below.*

$$\dfrac{\dfrac{\dfrac{\vdash P0}{Px, Qxy' \vdash P0}\,(Wk) \qquad Px, Qxy' \vdash Qxsy'}{Px, Qxy' \vdash P0 \wedge Qxsy'}\,(\wedge R)}{Px \wedge Qxy' \vdash P0 \wedge Qxsy'\ [*]}\,(\wedge L)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{Pu', Qu'su' \vdash Psu' \qquad Px, Qxy' \vdash Qxsy'}{Pu', Qu'su', Px, Qxy' \vdash Psu' \wedge Qxsy'}\,(\wedge R),(Wk)}{Pu' \wedge Qu'su', Px, Qxy' \vdash Psu' \wedge Qxsy'}\,(\wedge L)}{Pu' \wedge Qu'su', Px \wedge Qxy' \vdash Psu' \wedge Qxsy'\ [**]}\,(\wedge L)$$

It can be noticed that the leaves of the derivation are labeled with sequents formalizing ICs of the individual induction schemas for $Nx \vdash Px$ and $Nx, Ny \vdash Qxy$.

Proving the individual induction cases

The LKID$_a$ derivation for the IC '$(\Gamma \vdash \Delta)[\theta]$ attaches $S(N_l^1)[\delta^1] \dots S(N_l^p)[\delta^p]$' is given in Figure C.10a. Let p be the path used to build the cumulative substitution θ and leading the root N_r labeled by $(\Gamma \vdash \Delta)$ to the unique terminal node N_t in p. The rules applied along p can be used, as indicated by Lemma 9, to generate the sequent labeling N_t (resp., the node before N_t in p) from the sequent $\mathcal{C}(S(N_l^1))[\delta^1], \dots, \mathcal{C}(S(N_l^p))[\delta^p] \vdash \mathcal{C}(\Gamma \vdash \Delta)[\theta]$ formalising the IC, if p is $(Subst)$-free (resp., not $(Subst)$-free).

The LKID$_a$ proof is built according to the nature of N_t. If N_t is a leaf, p is $(Subst)$-free and the LKID$_a$ proof consists in the application of the 0-premise rule on $S(N_t)$. Let us assume that N_t is a bud having the node N_h as companion, for which $S(N_h) \equiv \Gamma_j \vdash \Delta_j$, and δ is i) $\sigma_{id}^{S(N_t)}$, if $S(N_t)$ is not the premise of a $(Subst)$ rule, or ii) the underlying substitution of the $(Subst)$-rule having $S(N_t)$ as premise, otherwise. If N_h is not from π, the LKID$_a$ proof for $S(N_t)$ is a (Wk)-step to get $\Gamma_j[\delta] \vdash \Delta_j[\delta]$, followed by the $(Subst)$-step to get $S(N_h)$. It finishes by the LKID$_a$ proof generated for $S(N_h)$ since N_h is $<_{\mathcal{R}}$-smaller than N_r. Finally, let us assume that N_h is from π. We recall that one can build the derivation of $\Gamma_j[\delta] \vdash \Delta_j[\delta]$ from $(\Gamma \vdash \Delta)[\theta]$, as explained by Lemma 9. Also, we remark that the same derivation is possible if its sequents have added in the antecedent part the same set of formulas, in particular $\mathcal{C}(S(N_l^1))[\delta^1], \dots, \mathcal{C}(S(N_l^p))[\delta^p]$ which is further represented as $\dots, \mathcal{C}(\Gamma_j[\delta] \vdash \Delta_j[\delta]), \dots$. Hence, $\dots, \mathcal{C}(\Gamma_j[\delta] \vdash \Delta_j[\delta]), \dots, \Gamma_j[\delta] \vdash \Delta_j[\delta]$ can be built from the IC $\dots, \mathcal{C}(\Gamma_j[\delta] \vdash \Delta_j[\delta]), \dots, \Gamma[\theta] \vdash \Delta[\theta]$. The LKID$_a$ proof for $S(N^t)$ starts by a (Wk)-step to get $\mathcal{C}(\Gamma_j[\delta] \vdash \Delta_j[\delta]), \Gamma_j[\delta] \vdash \Delta_j[\delta]$. It continues by deleting the implication, disjunction and conjunction operators using successive applications of $(\Rightarrow L)$, $(\wedge R)$, and $(\vee L)$, and finishes using (Ax) rules; this part is denoted by the horizontal dotted line from Figure C.10a.

Example 34 (cont. Example 33) *The LKID$_a$ proofs of the individual ICs are:*

$$\dfrac{}{\vdash P0}\,(R.(C.3)) \qquad \dfrac{\dfrac{}{Pu', Qu'su' \vdash Pu'}\,(Ax) \quad \dfrac{}{Pu', Qu'su' \vdash Qu'su'}\,(Ax)}{Pu', Qu'su' \vdash Psu'}\,(R.(C.4))$$

$$\dfrac{}{\vdash Qx0}\,(R.(C.5)) \qquad \dfrac{\dfrac{}{Px, Qxy' \vdash Px}\,(Ax) \quad \dfrac{}{Px, Qxy' \vdash Qxy'}\,(Ax)}{Px, Qxy' \vdash Qxsy'}\,(R.(C.6))$$

Building the LKID$_a$ proof of $\Gamma \vdash \Delta$

Displayed in Figure C.10c, it starts with a (Cut) rule, where the cut formula is the conjunction sequent for π. One branch is reduced to $\mathcal{C}(\Gamma \vdash \Delta), \Gamma \vdash \Delta$ which can be proved as for the last case discussed for Figure C.10a. The other branch is developed in Figure C.10b.

Example 35 (cont. Example 34) *The proof of $Nx, Ny \vdash Qxy$ starts by applying (Wk) followed by (Cut) with the conjunction sequent as cut formula:*

$$\dfrac{\dfrac{\vdash P0 \quad \vdash Qx0}{\vdash P0 \wedge Qx0}\,(\wedge R) \qquad \dfrac{\dfrac{Pu', Qu'su' \vdash Psu' \quad \dfrac{\vdash Qx0}{Pu', Qu'su' \vdash Qx0}\,(\mathrm{Wk})}{Pu', Qx'su' \vdash Psu' \wedge Qx0}\,(\wedge R)}{Pu' \wedge Qu'su' \vdash Psu' \wedge Qx0}\,(\wedge L) \qquad [*] \qquad [**]}{\vdash Pu \wedge Qxy}\,(Ind_a\ P_\pi)$$

Figure C.11: The beginning part of the LKID_a proof of $\vdash Pu \wedge Qxy$. The terminal nodes are individual ICs.

$$\dfrac{\dfrac{\textit{(cont. as in Figure C.11)}}{\vdash Pu \wedge Qxy} \quad \dfrac{\overline{Pu, Qxy \vdash Qxy}\,(Ax)}{Pu \wedge Qxy \vdash Qxy}\,(\wedge L)}{\vdash Qxy}\,(Cut)}{Nx, Ny \vdash Qxy}\,(Wk)$$

We give below some crucial properties of the conversion procedure.

Theorem 26 (termination) *The conversion procedure terminates. The set of new inductive predicate symbols and of the axioms defining them is finite.*

Proof Our arguments use the fact that the pre-proof tree-set given as input is finite. Hence, the number of iterations, given by the number of root nodes from the input proof, is finite. For each iteration, the operations in the case 1. of the procedure reproduce a tree derivation from the cyclic proof.

For the case 2., the construction of the explicit induction schema is an exhaustive combination of a finite number of individual induction schemas whose computation terminates. The conversion of the explicit induction proof terminates because the paths followed in the input proof are finite. □

Theorem 27 (soundness) *The output of the conversion procedure, applied on a CLKID$^\omega_N$ pre-proof tree-set in normal form and one of its root sequents S, is an LKID$_a$ proof of S if the new inductive predicates are admissible.*

Proof Let $(\mathcal{MD}, \mathcal{MR})$ be the input CLKID^ω_N proof. By Theorem 26, the conversion procedure builds for each sequent labeling a root node in $\mathcal{MD}$ a finite LKID_a-derivation whose terminal nodes are all leaves if the new inductive predicates are admissible. By Definition 15, these derivations are LKID_a proofs. Since S labels one of the roots of $\mathcal{MD}$, the conversion procedure builds an LKID_a proof of S. □

Time complexity

It has been shown in Subsection 7.2.3 that the normalization operation of CLKID^ω_N pre-proof is linear w.r.t. their number of nodes. Many steps in the execution of the conversion procedure consist in following paths from root to terminal nodes in the input CLKID^ω_N pre-proof tree-sets. The most costly operation stands in the combination of individual induction schemas during the generation of the axioms for the new admissible inductive predicates. Given a SCC with cycles π, if n is the number of its root nodes and m_i the number of ICs of the individual induction schema for the ith root node ($i \in [1..n]$), the number of axioms defining P_π is $m_1 \times \cdots \times m_n$. Each m_i ($i \in [1..n]$) is linear with the number of case applications from the tree rooted by the ith root node, which is bounded by the size k of the input cyclic proof. The number of SCCs with cycles is also bounded by k. Hence, the worst-case time complexity is k^n.

C.1.5 Checking the admissibility property

In the $\mathrm{FOL}_{\mathrm{ID}}$ setting, there are very few admissible inductive predicates, e.g., that defined by the unique axiom $P(x)$. Very simple and useful inductive predicates, as N, are not admissible.

Example 36 (N is not admissible in $\mathbf{FOL_{ID}}$) $\vdash N(a)$ *holds only if a is a natural number, i.e., of the form $s^n(0), n \geq 0$.*

However, N becomes admissible in a *typed* system where the type of the argument of N has the free constructors 0 and s. The idea is to use typed formalisms that can reproduce the cyclic and explicit induction reasoning from FOL_{ID}. An example of such a formalism is the *calculus of inductive constructions* [Paulin-Mohring, 2015, Bertot and Castéran, 2004], the logical framework behind Coq, because i) it accepts inductive definitions, and ii) every $\text{CLKID}^{\omega}_{N}$ and LKID_{a} inference rule can be simulated in Coq. The Coq inference system can directly reproduce the LK, equality, unfold and case rules. The explicit induction schema built by (Ind_a), when applied on an admissible inductive predicate P_π defined by axioms of the form (C.7), can be also reproduced by some recursive function that terminates and is complete, by using the functional programming style from Subsection 6.3.4.

In the following, we will show how to build such a function from cyclic pre-proofs.

Definition 17 (constrained $\text{CLKID}^{\omega}_{N}$ proof tree/tree-set) *A constrained $\text{CLKID}^{\omega}_{N}$ proof tree (resp.,tree-set) is every pre-proof tree (resp., tree-set) whose digraph has only n-cycles that discharge their IHs, according to Definition 12.*

Example 37 *We show that the pre-proof tree-set from Fig. C.9 can be constrained, too. Its digraph has one SCC with cycles denoted by π, with two 1-cycles and one 2-cycle. By using the cumulative substitutions built in Example 30, we define the ordering constraints for each n-cycle:*

- *for $[N^1, N^4, N^5, N^8, N^9]$ we have $S(N^8) <_\pi S(N^1)[\{x \mapsto x; y \mapsto sz; z \mapsto z\}]$;*
- *for $[N^{10}, N^{11}, N^{14}, N^{15}, N^{16}, N^{17}]$ we have $S(N^{16}) <_\pi S(N^{10})[\{x \mapsto sz; z \mapsto z\}]$;*
- *for $[N^1, N^4, N^5, N^7], [N^{10}, N^{14}, N^{15}, N^{18}, N^{19}]$ we have the two constraints $S(N^7) <_\pi S(N^1)[\{x \mapsto x; y \mapsto sz; z \mapsto z\}]$ and $S(N^{18}) <_\pi S(N^{10})[\{x \mapsto sz; z \mapsto z\}]$.*

We define the measure values for each sequent from the pre-proof tree-set, of the form $\ldots \vdash Pt$ (resp., $\ldots \vdash Qt_1t_2$), as the multiset $\{Nt, Nt\}$ (resp.,$\{Nt_1, Nt_1, Nt_2\}$).

The four ordering constraints are : $\{Nx, Nx, Nz\} <_\pi \{Nx, Nx, Nsz\}$, $\{Nz, Nz\} <_\pi \{Nsz, Nsz\}$, $\{Nx, Nx\} <_\pi \{Nx, Nx, Nz\}$, and $\{Nz, Nz, Nsz\} <_\pi \{Nsz, Nsz\}$. We can notice that the ordering and derivability constraints are satisfied if $<_a$ is defined as a rpo based on any precedence over the symbols N, 0 and s.

Lemma 13 *Let π be a SCC with cycles from the digraph of a constrained $\text{CLKID}^{\omega}_{N}$ proof tree-set, $F_\pi(\bar{t})$ its conjunction formula and P_π its new inductive predicate. If the principal formulas of the $(Case)$-steps are admissible inductive atoms, there is a constrained $\text{CLKID}^{\omega}_{N}$ proof tree of $\vdash P_\pi(\bar{t})$.*

Proof Let π be the SCC with cycles from the digraph of a constrained $\text{CLKID}^{\omega}_{N}$ proof tree-set $(\mathcal{MD}, \mathcal{MR})$ whose conjunction formula is $F_\pi(\bar{t})$. By construction, the axioms of P_π are issued from the individual induction schemas for the k (> 0) root sequents $S(N^{i_p})$ ($p \in [1..k]$) from π.

One can build a constrained $\text{CLKID}^{\omega}_{N}$ pre-proof tree of $\vdash P_\pi(\bar{t})$ by firstly building the pre-proof tree $\mathcal{D}$ of $\Gamma \vdash P_\pi(\bar{t})$, where $\Gamma \equiv \{p \mid p \text{ is an admissible IAA of } S(N^{i_p}), \forall p \in [1..k]\}$. For this, we establish a 'visiting' priority among the root sequents of π given, w.l.o.g., by the sequence $S(N^{i_1}), \ldots, S(N^{i_k})$. Let also $\bar{t}$ be $(\bar{t}_{i_1}, \ldots, \bar{t}_{i_k})$, where each $\bar{t}_{i_j}$ is $(\overline{t^1_{i_j}}, \ldots, \overline{t^{n_{i_j}}_{i_j}})$ ($j \in [1..k]$).

The process of building $\mathcal{D}$ is finished when the successive constructions of the *visiting path*, *unfolding* and *bud* parts are finished. The 'visiting path' part is built by successively considering each root sequent of π according to the fixed priority. Hence, we start by executing the rules that affect the IAAs along the paths leading N^{i_1} to each of the terminal nodes of the tree rooted by N^{i_1} in $\mathcal{MD}$, i.e., (Wk), $(contrL)$, (Gen) and $(Case)$. When finished, the sequent corresponding to a terminal node t is of the form $\Gamma' \vdash P_\pi(\overline{t^1_{i_1}[\theta_c]}, \ldots, \overline{t^{n_{i_1}}_{i_1}[\theta_c]}, \bar{t}_{i_2}, \ldots, \bar{t}_{i_k})$, where θ_c is the cumulative substitution for the path p'_{i_1}, leading N^{i_1} to t, and Γ' is Γ for which the admissible IAAs of $S(N^{i_1})$ are replaced by those of the sequent resulting after the last application of a rule that generates new admissible IAAs, i.e., by the last application of (Gen) in p'_{i_1}. We continue to build $\mathcal{D}$ by considering N^{i_2} and using some path p'_{i_2}, as it has been done for N^{i_1} and p'_{i_1}. This part finishes when the derivation for the last root sequent in the sequence, N^{i_k}, was built. At the end, the succedent atom from each resulting sequent is the conclusion

of some axiom defining P_π (modulo variable renaming).

The unfolding part applies the unfold rules with the axioms defining P_π.

The bud part consists in the application of $(Wk)/(contrL)$ to delete/duplicate IAAs in order to successfully apply $(Subst)$ and generate buds. Let $(Subst)$ be applied on a sequent S of the form $\Gamma \vdash P_\pi(\overline{t''})$, where $P_\pi(\overline{t''})$ is the condition of some axiom defining P_π (modulo variable renaming). Then the substitution underlying $(Subst)$ is the composition of the substitutions δ used to define the IHs that helped to build the condition $P_\pi(\overline{t''})$ when defining the axioms of P_π. Let $\mathcal{R}$ be the induction function assigning the root as companion of every bud node.

We show that the pre-proof tree $(\mathcal{D}, \mathcal{R})$ is a constrained proof tree. $\mathcal{D}$ has only one SCC with cycles, let it be denoted by π', including only 1-cycles represented by paths leading the root to each of its buds. Every bud sequent is a premise of some $(Subst)$ rule, so the IHs to be discharged by the 1-cycles are the conclusions of these rules. Let $[N, \ldots, N^f, B]$ be such a 1-cycle. In order to show that the IH $S(N^f)$ is discharged by this 1-cycle, we define for any arbitrary substitution θ the measure value for the instance $(\Gamma \vdash P_\pi(\bar{t}_{i_1} \ldots, \bar{t}_{i_k}))[\theta]$ of $S(N)$ as the multiset $\{A_{S(N^{i_1}[\theta]}, \ldots, A_{S(N^{i_k}[\theta]}\}$. According to Definition 12, it is sufficient to show that $S(N^f)$ is $\ll_{\pi'}$-derivable from $S(N)[\theta'_c]$ along $[N, \ldots, N^f, B]$, where θ'_c is the cumulative substitution for the path $[N, \ldots, N^f, B]$. Since $N = \mathcal{R}(B)$, we have $S(N^f) \equiv S(N)[\delta]$, where δ is the substitution underlying the $(Subst)$ rule that has $S(B)$ as premise. Therefore, it is enough to show that $S(N)[\delta]$ is $\ll_{\pi'}$-derivable from $S(N)[\theta'_c]$ along $[N, \ldots, N^f, B]$. This condition holds because, for each $j \in [1..k]$, there is $l \in [1..k]$ chosen from an IC, of the form '$S(N^{i_l})[\theta'_c]$ attaches $\{\ldots, S(N^{i_j})[\delta], \ldots\}$', from some individual IC used during the explicit induction proof requiring P_π such that $A_{S(N^{i_j})[\delta]}$ is $<_{\pi'}$-derivable from $A_{S(N^{i_l})[\theta'_c]}$, where $<_{\pi'}$ is defined as $<_\pi$. The derivability conditions from the $<_{\pi'}$-derivability relation hold thanks to the way $\mathcal{D}$ was build during the 'visiting path' part.

Finally, the derivation tree of $\vdash P_\pi(\bar{t})$ is built by successively applying (Cut) on every IAA p from Γ. Since p is admissible, (Adm) can be applied on the sequent $\vdash p, P_\pi(\bar{t})$. At the end, we get $\Gamma \vdash P_\pi(\bar{t})$ that is a bud. Since no new n-cycle is built, the pre-proof tree-set consisting of the two derivation trees rooted by $\vdash P_\pi(\bar{t})$ and $\Gamma \vdash P_\pi(\bar{t})$ is a constrained CLKID$^\omega_N$ proof tree. □

Example 38 *The pre-proof tree from Fig. C.12 is a constrained CLKID$^\omega_N$ proof tree $(\mathcal{D}, \mathcal{R})$, too. (Gen) denotes the restricted form of $(= L)$ used by CLKID$^\omega_N$.*

We have to check that their IHs are discharged along the four 1-cycles of the unique SCC π'. It is sufficient to check that:

- $A_{Nx,Ny',Nx \vdash P_\pi xxy'}$ *is $\ll_{\pi'}$-derivable from the cumulative instance $A_{S(\dagger)[\{u \mapsto 0; x \mapsto x; y \mapsto sy'; y' \mapsto y'\}]}$ along $[\ddagger, \ldots, \ddagger 1]$. This holds since the two IAA multisets of $A_{Nx,Nx,Ny' \vdash P_\pi xxy'}$, i.e., $\{Nx, Nx\}$ and $\{Nx, Nx, Ny'\}$, are $<_{\pi'}$-derivable from the IAA multiset $\{Nx, Nx, Nsy'\}$ of $A_{S(\ddagger)[\{u \mapsto 0; x \mapsto x; y \mapsto sy'; y' \mapsto y'\}]}$ along $[\ddagger, \ldots, \ddagger 1]$;*
- $A_{Nu',Nu',Nsu' \vdash P_\pi u'u'su'}$ *is $\ll_{\pi'}$-derivable from the cumulative instance $A_{S(\ddagger)[\{u \mapsto su'; u' \mapsto u'; x \mapsto x; y \mapsto 0\}]}$ along $[\ddagger, \ldots, \ddagger 2]$. This is true because $\{Nu', Nu'\}$ and $\{Nu', Nu', Nsu'\}$ are $<_{\pi'}$-derivable from $\{Nsu', Nsu'\}$ along $[\ddagger, \ldots, \ddagger 2]$;*
- $A_{Nu',Nu',Nsu' \vdash P_\pi u'u'su'}$ *is $\ll_{\pi'}$-derivable from the cumulative instance $A_{S(\ddagger)[\{u \mapsto su'; u' \mapsto u'; x \mapsto x; y \mapsto sy'; y' \mapsto y'\}]}$ along $[\ddagger, \ldots, \ddagger 3]$. This also holds because $\{Nu', Nu'\}$ and $\{Nu', Nu', Nsu'\}$ are $<_{\pi'}$-derivable from $\{Nsu', Nsu'\}$ along $[\ddagger, \ldots, \ddagger 3]$;*
- *finally, $A_{Nx,Nx,Ny' \vdash P_\pi xxy'}$ is $\ll_{\pi'}$-derivable from the cumulative instance $A_{S(\ddagger)[\{u \mapsto su'; u' \mapsto u'; x \mapsto x; y \mapsto sy'; y' \mapsto y'\}]}$ along $[\ddagger, \ldots, \ddagger 4]$. This is true because $\{Nx, Nx\}$ and $\{Nx, Nx, Ny'\}$ are $<_{\pi'}$-derivable from $\{Nx, Nx, Nsy'\}$ along $[\ddagger, \ldots, \ddagger 4]$.*

Well-founded proof tree-sets and implementation in Coq The well-founded proof trees/tree-sets are similar to the constrained ones, excepting that there are no derivability constraints and that the orderings over literals and formulas should be well-founded. Such orderings have been successfully

$$\dfrac{\dfrac{\dfrac{}{\vdash P_\pi 0x0}\,(R.(C.8))}{y=0\vdash P_\pi 0xy}\,(Gen) \quad \dfrac{\dfrac{\dfrac{Nu,Nx,Ny\vdash P_\pi uxy\ \textbf{(‡1)}}{Nx,Nx,Ny'\vdash P_\pi xxy'}\,(Subst)}{Nx,Ny'\vdash P_\pi xxy'}\,(contrL)}{y=sy',Nx,Ny'\vdash P_\pi 0xy}\,(R.(C.9))}{Nx,\underline{Ny}\vdash P_\pi 0xy}\,(Gen)}{u=0,Nx,Ny\vdash P_\pi uxy}\,(Case\ N) \quad \dfrac{\dfrac{\dfrac{\dfrac{Nu,Nx,Ny\vdash P_\pi uxy\ \textbf{(‡2)}}{Nu',Nu',Nsu'\vdash P_\pi u'u'su'}\,(Subst)}{Nsu',Nu'\vdash P_\pi u'u'su'}\,(contrL)}{Nu',Nsu',Nx\vdash P_\pi u'u'su'}\,(Wk)}{y=0,Nu',Nsu',Nx\vdash P_\pi su'x0}\,(R.(C.10))}{Nu',Nsu',Nx,\underline{Ny}\vdash P_\pi su'xy}\,(Gen)}{Nu,u=su',Nu',Nx,Ny\vdash P_\pi uxy}\,[*]\,(Case\ N)$$

$$\dfrac{\dfrac{\dfrac{Nu,Nx,Ny\vdash P_\pi uxy\ \textbf{(‡3)}}{Nu',Nu',Nsu'\vdash P_\pi u'u'su'}\,(Subst)}{Nu',Nsu'\vdash P_\pi u'u'su'}\,(contrL) \quad \dfrac{\dfrac{Nu,Nx,Ny\vdash P_\pi uxy\ \textbf{(‡4)}}{Nx,Nx,Ny'\vdash P_\pi xxy'}\,(Subst)}{Ny',Nx\vdash P_\pi xxy'}\,(contrL)}{Nu',Nsu',Nx,Ny'\vdash P_\pi u'u'su'}\,(\mathrm{Wk}) \quad \dfrac{}{Nu',Nsu',Nx,Ny'\vdash P_\pi xxy'}\,(\mathrm{Wk})}{Nu',Nsu',Nx,Ny'\vdash P_\pi su'xsy'}\,(R.(C.11))}{y=sy',Nu',Nsu',Nx,Ny'\vdash P_\pi su'xy\ [*]}\,(Gen)$$

(a) The derivation tree of $Nu, Nx, Ny \vdash P_\pi uxy$.

$$\dfrac{\dfrac{}{\vdash Nv,P_\pi uxy}\,(Adm\ Nv) \quad \dfrac{\dfrac{}{\vdash Nx,P_\pi uxy}\,(Adm\ Nx) \quad \dfrac{\dfrac{}{\vdash Nu,P_\pi uxy}\,(Adm\ Nu) \quad Nu,Nx,Nv\vdash P_\pi uxy\ (‡)}{Nx,Nv\vdash P_\pi uxy}\,(Cut)}{Nv\vdash P_\pi uxy}\,(Cut)}{\vdash P_\pi uxy}\,(Cut)$$

(b) The derivation tree of $\vdash P_\pi uxy$.

Figure C.12: The pre-proof tree-set for showing the admissibility of $P_\pi uxy$.

used in Chapter 6 to certify cyclic induction reasoning in Coq. The same approach has been adopted to implement the termination orderings for the new inductive predicates, based on i) the Coccinelle library to associate syntactical weights to Coq formulas and to define Noetherian and 'stable under substitutions' orderings over formulas from term algebras, ii) the CoLoR library to build the multiset extensions of the orderings defined with Coccinelle, and iii) the functional (explicit induction) schemas.

Example 39 (Coq proof of the P&Q example by explicit induction) *Compared to Example 1, we give definitions of the P and Q inductive predicates that accept only naturals as arguments:*

```
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).
```

The termination ordering annotates the definition of P_π using the axioms from Example 32. For convenience, the triplets (x, y, z) given as argument to P_π are represented with pairs as $(x, (y, z))$, for any natural x, y, z:

```
Function Pπ (a: nat × (nat × nat)) {wf (fun u v: nat × (nat × nat) ⇒ match u,v with (u1, (x1,
y1)), (u2, (x2, y2)) ⇒ mless [[(model_nat u1), (model_nat u1)], [(model_nat x1), (model_nat
x1), (model_nat y1)]] [[(model_nat u2), (model_nat u2)], [(model_nat x2), (model_nat x2),
(model_nat y2)]] end) a}: Prop :=
   match a with
  | (0, (_, 0)) ⇒ True
  | (0, (x', (S y'))) ⇒ Pπ (x', (x', y'))
  | ((S u'), (_, 0)) ⇒ Pπ (u', (u', (S u')))
  | ((S u'), (x', (S y'))) ⇒ Pπ (u', (u', (S u'))) ∧ Pπ (x', (x', y'))
end.
```

where (model_nat x) is the Coccinelle representation for the variable x of sort **nat** *and* mless *is the multiset extension of an ordering over multisets of terms. The measure value for the atom Pt (resp., Qt_1t_2) is the multiset $\{t, t\}$ (resp.,$\{t_1, t_1, t_2\}$), for any terms t, t_1, and t_2. The user should provide the termination proof that checks whether the argument a decreases after each call w.r.t. the ordering following the* `wf` *keyword.*

P_π is also complete because all possible cases for its argument a are considered by the `match` *construction. It helps to define the explicit induction schema P_π_ind as a new functional schema:*

```
Functional Scheme Pπ_ind := Induction for Pπ Sort Prop.
```

The inductive predicate **PQ** *defines the conjunction of their root formulas, which are simplified by taking into account the logical representation of sequents and that N is admissible:*

```
Inductive PQ: nat × (nat × nat) → Prop :=
  r0: ∀ u x y: nat, (P u) ∧ (Q x y) → PQ (u, (x, y)).
```

The application of P_π_ind will start the proof of **PQ** `(u, (x, y))`*:*

```
Theorem pq_is_true : ∀ u x y, PQ (u, (x, y)).
Proof.
  intros.
   (* application of the explicit induction schema *)
  pattern (u, (x, y)), (Pπ (u, (x, y))). apply Pπ_ind; intros; apply r0; subst.
   (* case P 0 ∧ Q x 0 *)
```

```
- split. apply p0. apply q0.
  (* case P 0 ∧ Q x' (S y') *)
- inversion H. subst. destruct H1. inversion H1. subst.
    split. apply p0. apply q1. trivial. trivial.
    split. apply p0. inversion H. subst. destruct H7. apply q1. trivial. trivial.
  (* case P (S u') ∧ Q _x 0 *)
- inversion H. subst. destruct H1. split. apply p1. trivial. trivial. apply q0.
  (* case P (S u') ∧ Q x'0 (S y') *)
- inversion H. subst. inversion H0. subst. destruct H2. destruct H3.
  split. apply p1. trivial. trivial. apply q1. trivial. trivial.
Qed.
```

After the unfolding of **PQ** *in the goal of each induction case, we get only conjunctions. The proof of each of their conjuncts, resulting after the application of* `split`*, is built by firstly unfolding with* `apply` *the definitions of* **P** *and* **Q***, as shown in Figure 4. After the use of the* `inversion` *and* `destruct` *tactics on the IHs defined by the induction schema, the remaining formulas are proved by* `trivial` *because they also occur in the premise part of the goal.*

Finally, the main theorem is proved as a conjunct of the previous theorem:

```
Theorem q_is_true : ∀ x y, Q x y.
Proof.
  intros.
  assert (H:= pq_is_true x x y). inversion H. destruct H1. trivial.
Qed.
```

C.1.6 Certifying other cyclic proofs

	Conjecture	# SCC	Root sequents	Measure values
1.	$N_1(x) \vdash O(x) \vee E(x)$	1	$N_1(x) \vdash O(x) \vee E(x)$	$\{x\}$
2.	$N_1(x) \vdash Add(x, 0, x)$	1	$N_1(x) \vdash Add(x, 0, x)$	$\{x\}$
3.	$N_1(x) \wedge N_2(y) \wedge Add_3(x, y, z) \vdash Add(x, s(y), s(z))$	1	$N_1(x) \wedge N_2(y) \wedge Add_3(x, y, z) \vdash Add(x, s(y), s(z))$	$\{x, z\}$
4.	$N_1(x) \wedge N_2(y) \vdash R(x, y)$	2	$N_1(x) \wedge N_2(y) \vdash R(x, y)$ $N_1(w) \vdash R(s(w), 0)$	$\{y\}$ $\{w\}$
5.	$N_1(x) \wedge N_2(y) \vdash p(x, y)$	1	$N_1(x) \wedge N_2(y) \vdash p(x, y)$	$\{x, y\}$
6.	$(memberT(x,insIn(y, z, t)) = true \wedge memberT(x, y) = false) \rightarrow z = x$	0		

Table C.5: Statistics about some cyclic proofs.

Table C.5 gives statistics about the cyclic induction proofs of some relevant conjectures taken from the repository of CYCLIST (Conjectures 1 to 4), the 2-Hydra example [Berardi and Tatsuta, 2019] (Conjecture 5), and the repository of SPIKE (Conjecture 6), by showing the number of SCCs and their root sequents with the measure values to be used in the definition of the induction ordering. We will discuss the Coq definitions of P_π, proposed by the conversion procedure for each conjecture. Only the match cases for the argument a of P_π are illustrated.

```
| 0 ⇒ True
| (S y) ⇒ Pπ y
```

```
| (0, (_ , _)) ⇒ True
| ((S x), (_, 0)) ⇒ True
| ((S x), (u, (S y))) ⇒ Pπ (x, (u, y))
```

The lhs P_π definition from above was used in the proof of Conjecture 1 involving the even (E) and odd (O) predicates, Conjecture 2 concerning the addition (Add), and the formula corresponding to the first root sequent required by Conjecture 4. Being the simplest one, `apply` P_π`_ind` is equivalent to apply the Peano principle. The rhs definition of P_π from above was used to prove Conjecture 3.

The lhs definition from below helped to prove the other root sequent for Conjecture 4. The rhs one was used to prove the 2-Hydra example. Both are complex and arguably hard to be found eagerly.

```
| (0, _) | ((S 0), 0) | ((S (S 0)), 0) | ((S (S (S _))), 0) ⇒ True
| ((S 0), (S z)) ⇒ Pπ ((S (S 0)), z)
| ((S (S 0)), (S z)) ⇒ Pπ ((S (S (S 0))), z)
| ((S (S (S y))), (S z)) ⇒ Pπ ((S (S (S (S y)))), z)
```

```
| (0, 0) | ((S 0), 0) | (_, (S 0)) ⇒ True
| ((S (S x)), 0) ⇒ Pπ ((S x), x)
| (0, (S (S y))) ⇒ Pπ ((S y), y)
| ((S x), (S (S y))) ⇒ Pπ (x, y)
```

The last cyclic proof, requiring 8 induction steps, is the largest one. It was built by SPIKE for the conjecture `member_t_insin` from Table 2.1, one of the 79 conjectures that helped to prove the equivalence between two conformity algorithms for the ATM networks. It will not be presented here but it can be accessed, as the other proofs and experimental material, from https://drive.google.com/file/d/10HUmH4ywl1O1PSPjODH58zXwL6K_KBeL/view?usp=sharing

C.1.7 Further lines of research

We showed that the use of types allows to define admissible inductive predicates and the application of the conversion procedure in logics that can reproduce the FOL_{ID} inductive reasoning, e.g., the calculus of inductive constructions. This allowed us to certify the FOL_{ID} cyclic reasoning with Coq.

We also showed in Example 2 that Coq is able to build cyclic pre-proofs. We intend to build a *Coq plugin* that validates such pre-proofs. It can be imagined that the user activates a mode before starting the construction of the pre-proofs such that, when the construction process finishes with 'Qed.', the validation of the pre-proofs is executed.

The validation step can be done in two ways, by using:

1. formula-based Noetherian induction principles, as shown for the cyclic formula-based Noetherian induction pre-proofs from Chapter 6. Recall that the Coq certification process of implicit induction proofs can be completely automatized. Since the pre-proofs can be represented as formula-based Noetherian induction pre-proofs, we expect that the validation step be performed automatically, too;

2. term-based Noetherian induction principles, as shown in this subsection. We expect that the validation process be also highly automatic.

The plugin may include minimal user interactions for choosing the right induction orderings or finishing the termination proofs of the new inductive predicates/functions.

We also plan to certify FOL_{ID} cyclic pre-proofs based on a validation process using formula-based Noetherian induction principles. We illustrate below our approach for the CYCLIST pre-proof of the main P&Q conjecture.

The Coq script. The CYCLIST specification of the main P&Q conjecture and its pre-proof from Figure 7.6 can be translated in the following Coq script:

```
Theorem true_0: ∀ x y, N x → N y → Q x y.
Proof.
intros x y H H0.
(* instantiate y from Q x y *)
inversion H0. rewrite ← H1.
(* Q x zero *) apply q1.
rename x0 into z.
(* Q x (succ z) *) apply q2. split.
(* the proof of Q x z requires induction *)
Focus 2. (* code follows on the next column *)
```

```
(* instantiate x from P x *)
inversion H. rewrite ← H3.
(* P zero *) apply p1.
clear y H0 H2. rename x0 into y.
 (* P (succ y) *) apply p2.
assert (Q y y ∧ P y).
(* the proof of Q y y ∧ P y needs induction
*)
Focus 2.
```

The pre-proof ends by showing the equivalence of **Q** *y y* ∧ **P** *y* and **Q** *y y* ∧ **P** (*succ y*):

```
destruct H0. split; trivial. apply q2. split; trivial.
```

Admitted.

The Coq certification. Following the certification methodology from Chapter 6, one should firstly define the mpo and its multiset extension denoted by `less`. According to the Coccinelle specification of the mpo, to each function symbol from the Coq specification corresponds a Coccinelle function symbol with an arity, status and index :

```
Inductive symb : Set :=
| id_0
| id_S
.

Definition arity (f:symb) :=
  match f with
  | id_0 ⇒ term_spec.Free 0
  | id_S ⇒ term_spec.Free 1
  end.

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

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

The function *model_nat* translates Coq terms to Coccinelle terms:

```
Variable model_nat : T → term.
Axiom model_nat_0: model_nat zero = (Term id_0 nil).
Axiom model_nat_succ: ∀ (u1: T), model_nat (succ u1) = (Term id_S ((model_nat u1)::nil)).
```

The above equalities can be used for rewriting during the proofs:

```
Hint Rewrite model_nat_0 model_nat_succ : model_nat.
Ltac rewrite_model := autorewrite with model_nat.
```

If the pre-proof has several non-singleton SCCs, one should define a precedence w.r.t. their processing order, such that the SCC S_1 is treated after the SCC S_2 if S_1 depends on S_2. For each SCC S,

1. attach a measure value to each root from S inside a *functional* to ensure that whenever the root formula is instantiated the mesure value is also instantiated with the same substitution:

   ```
   Definition type_LF_0_4 := T → T → (Prop × (List.list term)).

   Definition F_0 : type_LF_0_4:= (fun u1 u2 ⇒ (N u1 → N u2 → Q u1 u2, ((model_nat u1)::(model_nat u1)::(model_nat u2)::(model_nat u2):: nil))).
   Definition F_4 : type_LF_0_4:= (fun u1 u2 ⇒ (N u1 → N u2 → N (succ u2) → Q u1 u2 ∧ P u1, ((model_nat u1)::(model_nat u1)::(model_nat u2)::(model_nat u2):: (model_nat (succ u2))::nil))).
   ```

2. define the list of functionals:

   ```
   Definition LF_0_4 := [F_0,F_4].
   ```

3. define and prove the main lemma by reproducing the parts rooted by the root nodes in the pre-proof; the buds are validated using well-founded induction arguments:

   ```
   Lemma main_0_4 : ∀ F, In F LF_0_4 → ∀ u1, ∀ u2, (∀ F', In F' LF_0_4 → ∀ e1, ∀ e2, less (snd (F' e1 e2)) (snd (F u1 u2)) → fst (F' e1 e2)) → fst (F u1 u2).
   ```

4. prove all formulas from the list of functionals using a formula-based instance of the well-founded induction principle [Stratulat, 2012] as the all_true conjecture:

   ```
   Theorem all_true_0_4: ∀ F, In F LF_0_4 → ∀ u1: T, ∀ u2: T, fst (F u1 u2).
   ```

5. finally, prove the root conjectures as direct consequences of all_true:

Theorem true_0: $\forall\ x\ y,\ \mathbf{N}\ x \to \mathbf{N}\ y \to \mathbf{Q}\ x\ y.$

Theorem true_4: $\forall\ x\ y,\ \mathbf{N}\ x \to \mathbf{N}\ y \to \mathbf{N}\ (succ\ y) \to \mathbf{Q}\ x\ y \wedge \mathbf{P}\ x.$

C.2 Other projects

C.2.1 Strategies for directly building sound CLKID$^{\omega}_{N}$ pre-proofs

We are interested to develop strategies for building sound CLKID$^{\omega}_{N}$ pre-proofs without backtracking, which may lead to faster proof developments. A first attempt would be to use Theorem 19. It suggests that sequents can also be proved by directly building sound pre-proof tree-sets. For this purpose, we adapt the DRaCuLa strategy from Chapter 2. Mainly, the trees from a pre-proof tree-set are developed by applying the CLKID$^{\omega}_{N}$ rules, as usual, with the following exceptions:

- when applying a (*Subst*)-rule, the premise becomes a bud sequent, as shown for the first transformation of the normalization procedure. The next step is to develop a new tree rooted by the companion of the bud;

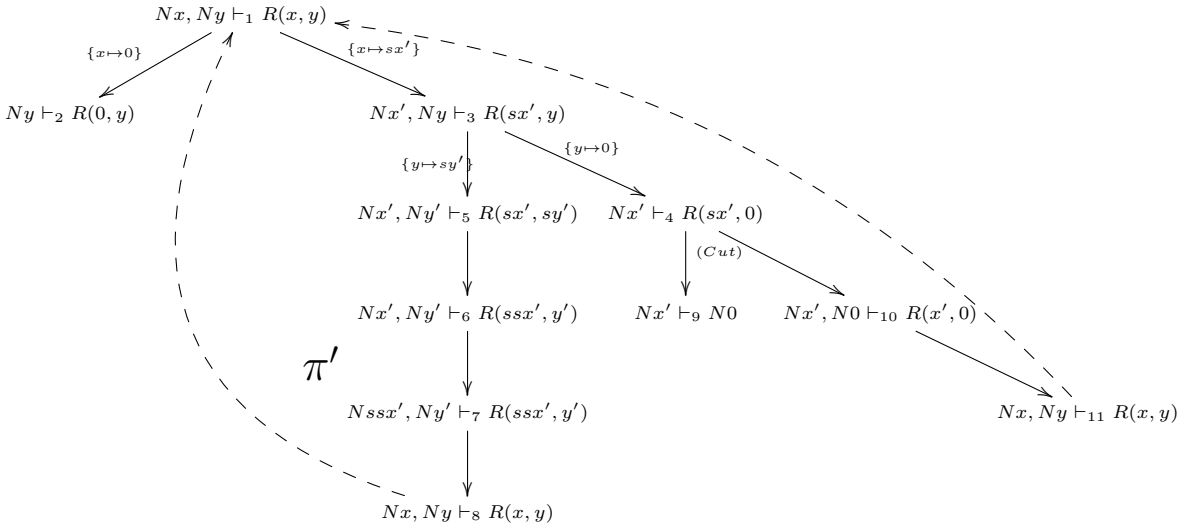
- when a bud is about to be created, several scenarios may happen. As a preliminary step, if its companion is a non-root node, the second transformation of the normalization procedure is applied. If the bud candidate is part of a SCC that discharges its IHs, the bud is created (scenario 1). If it is not yet the case, either i) the strategy tries to build a non-singleton SCC, by developing parts from other trees (scenario 2), or ii) the SCC does not discharge its IHs (scenario 3); in this case, a backtracking step is required either to redefine the ordering at the SCC level, or to redo previous steps, or to continue to develop the proof by applying a CLKID$^{\omega}_{N}$ rule on the sequent labelling the bud candidate.

  For (scenario 1), not only the current bud candidate is created, but all the bud candidates from the SCC are built, hence *simultaneous* induction is performed.

Example 40 *The above strategy can build the pre-proof tree from Example 23. The progression in its construction can be retraced by following the indexes of the sequents labelling the nodes from the digraph displayed in Example 25.*

This proof strategy uses heuristics based on ordering constraints, different from the iterative depth-first search heuristics used by Cyclist. The induction ordering for each non-singleton SCC should be defined only when its construction is completely finished. This is because the induction orderings used to partially discharge some of the IHs may not be sufficient to discharge the new IHs occurring in the pre-proof. If no ordering is found, backtracking steps may be required.

Example 41 *One could have built a new bud of (*) from the pre-proof tree of Example 23, by developing* (†) *such that* $N0$ *is added as IAA, then* (*Subst*) *applied conveniently. The new non-singleton SCC from its digraph is part of the SCC* π' *of the digraph from Example 25. However, the induction ordering* $<_{\pi'}$*, defined in Example 26 and used to discharge the IH* N^7*, cannot be used to discharge the IH* N^{10}*. Hence, a different induction ordering is required, if any, to discharge the IHs of the new SCC* π'*.*



In order to get rid of the backtracking, we can establish a link with reductive reasoning techniques [Stratulat, 2017a]. The idea is that sound pre-proof trees can also be directly generated to satisfy implicitly the ordering constraints, similar to implicit induction proofs, by using a *reductive* proof strategy based on a unique induction ordering $<$. Such strategy guarantees that, for every two successive nodes N^i and N^{i+1} from each path p, of the form $[N^1, \ldots, N^n]$ and occurring in the definition of some minimal cycle of its digraph, and $i \in [1..f-1]$, we have either $A_{S(N^{i+1})[\theta^c_{i+1}]} \equiv A_{S(N^i)[\theta^c_i]}$ or $S(N^{i+1})[\theta^c_{i+1}]$ is $<$-derivable from $S(N^i)[\theta^c_i]$ along p, where f is the index of the IH-node in $[N^1, \ldots, N^n]$ and θ^c_j is the cumulative substitution for the path $[N^j, \ldots, N^f]$ $(j \in [1..f])$. The derivability constraints are satisfied if the syntactic equality relation is not satisfied at least once along p. Indeed, knowing that the $<$-derivability relation is transitive (Lemma 11), we have that $S(N^f)$ is $<$-derivable from $S(N^1)[\theta^c_1]$ along p, as required. If the rule applied at step i is different from $(= L)$, we have that $\theta^c_i \equiv \theta^c_{i+1}$. In this case, it is sufficient to ensure instead that $A_{S(N^{i+1})} \equiv A_{S(N^i)}$ or $S(N^{i+1})$ is $<$-derivable from $S(N^i)$ along p, due to the 'stability under substitutions' property of $<$-derivability (again Lemma 11).

Example 42 *As a proof of concept, we define the derived rule* $(DCase)$*:*

$$\frac{S_1 \quad \ldots \quad S_n}{\Gamma, P(\overline{x}) \vdash \Delta}\,(DCase\ P) \qquad as \qquad \frac{\dfrac{\dfrac{S_1}{case\ distinction}\,(= L) \quad \ldots \quad \dfrac{S_n}{case\ distinction}\,(= L)}{\Gamma, P(\overline{x}), \underline{P(\overline{x})} \vdash \Delta}\,(Case\ P)}{\Gamma, P(\overline{x}) \vdash \Delta}\,(contr L)$$

where $\overline{x}$ *is a vector of variables. We also define the* (Bud) *rule:*

$$\frac{(bud\ sign)}{\Gamma \vdash \Delta}\,(Bud) \qquad as \qquad \frac{\dfrac{\Gamma' \vdash \Delta'\ (bud\ sign)}{\Gamma'[\sigma] \vdash \Delta'[\sigma]}\,(Subst)}{\Gamma \vdash \Delta}\,(Wk)$$

if $\Gamma' \vdash \Delta'$ subsumes $\Gamma \vdash \Delta$ *with substitution* σ*, i.e.,* $\Gamma'[\sigma] \subseteq \Gamma$ *and* $\Delta'[\sigma] \subseteq \Delta$*. Different variants of the subsumption operation are widely employed by the current theorem provers,* CYCLIST *being one of them.*

By using the alternative notation without parentheses, a pre-proof of $Nx, Ny \vdash Q(x,y)$ *is built below by firstly trying to apply the unfold rules followed by* (Bud)*, then* (Del) *and, finally,* $(DCase)$*.* (Del) *is a restricted version of the* (Wk) *rule that deletes the IAAs of the form* $N(t)$ *if none of the inductive succedent atoms from the conclusion has* t *as argument. It can be noticed that the history of every IAA occurring in each premise of any rule* r *from the above rules but* (Bud) *has one of the IAAs from the conclusion of* r*.*

$$\dfrac{\dfrac{}{N0 \vdash P0}\,(R.(7.1)) \quad \dfrac{\dfrac{\dfrac{(*1)}{Nsz, Nz \vdash Pz}\,(Bud) \quad \dfrac{(\dagger 2)}{Nz, Nsz \vdash Qzsz}\,(Bud)}{Nsz, Nz \vdash Psz}\,(R.(7.2))}{Nx \vdash Px\ (*)}\,(DCase\ N)}{Nx, Nz, Nsz \vdash Px}\,(Del) \quad \dfrac{(\dagger 1)}{Nx, Nz, Nsz \vdash Qxz}\,(Bud)}{\dfrac{Nx, N0 \vdash Qx0}{Nx, Nz, Nsz \vdash Qxsz}\,(R.(7.3)) \quad (R.(7.4))}\,(DCase\ N)}{Nx, \underline{Ny} \vdash Qxy\ (\dagger)}$$

The proof strategy is reductive if the measure value for each sequent of the form $\Gamma, N(t) \vdash P(t)$ *(resp.,* $\Gamma, N(t_1), N(t_2) \vdash Q(t_1, t_2)$ *is the multiset of IAAs* $\{N(t), N(t)\}$ *(resp.,* $\{N(t_1), N(t_1), N(t_2)\}$*) and* $<$ *is defined as the multiset extension of the ordering* $<_a$ *over IAAs from Example 25. It can be checked that the* $<$*-derivability constraints are satisfied, by taking into account that the unique non-singeton SCC of the digraph associated to its normalized pre-proof tree-set includes the rb-paths* $[(*), \ldots, (*1)]$*,* $[(\dagger), \ldots, (\dagger 1)]$*,* $[(*), \ldots, (\dagger 2)]$*, and* $[(\dagger), \ldots,$ *copy of* $(*)]$*.*

By Theorem 19, our approach allows to prove several conjectures *simultaneously*. This is a feature specific to formula-based Noetherian induction reasoning which has been already employed by the implicit induction inference systems in Subsection 6.2. It is particularly useful when the proofs of the conjectures are mutually dependent.

Example 43 *The normalization step for the pre-proof tree from Example 42 can be avoided if the pre-proof trees of* $Nx \vdash Px$ *and* $Nx, Ny \vdash Qxy$ *are developed simultaneously.*

The ordering constraints are implicitly satisfied by reductive proof strategies.

In the future, we plan to define new (derived) rules and proof strategies that *automatically* generate more compact reductive proof derivations and provide a better control of the proof development. We intend to integrate these strategies in E-Cyclist. The main challenge of our approach remains to find the 'good' induction orderings.

C.2.2 Certification of saturation-based proofs

In a future line of research, we intend to exploit the link established between Noetherian induction reasoning and saturation-based reasoning in order to apply our certification methodologies from Chapter 6 for certifying the inductionless induction reasoning and other saturation-based reductive reasoning, as those described in Chapter 8.

C.2.3 Applications

A. Certification of the algorithms issued from algorithm synthesis

Our experiments in the *Theorema* system from Section 3.2 show how one can discover numerous algorithms for the same functions, differing in efficiency and complexity if one applies different induction principles and chooses different alternatives in the proofs. The Coq certification of the synthesized algorithms can be seen as a test for checking the soundness of our approach.

An alternative to avoid the certification step is the generation of the proofs and the implementation of the inference rules and strategies directly in Coq. This would ensure that every synthesized algorithm by these inference rules and strategies is implicitly sound. Some downsides of this approach are: i) its difficulty to prove the soundness of the inference system, and ii) the inadequacy for rapid prototyping and testing new ideas. A reasonable compromise would be to devise procedures for translating the *Theorema* proofs directly into *Coq* scripts, by following similar translation procedures as those used for implicit induction proofs in Subsection 6.2.

B. JavaCard

Further case studies. It would be interesting to provide automatic support for the construction and cross-validation of the virtual machines for properties other than typing, e.g. initialization or non-

interference. As such efforts rely on similar methodologies described in Chapter 5, it seems interesting to use them to prove the appropriate versions of the CDO, CDA and MON properties.

Domain-specific proof environment for certifying low-level languages. A longer term objective would be to develop an environment that provides automatic support for certifying low-level languages. We expect the restricted format of JSL will prove useful for this task.

Bibliography

[Aczel, 1977] P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North Holland, 1977.

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

[Aoto and Stratulat, 2014] T. Aoto and S. Stratulat. Decision procedures for proving inductive theorems without induction. In *Proc. of the 16th International Symposium on Principles and Practice of Declarative Programming (PPDP 2014)*, pages 237–248. ACM Press, 2014.

[Aoto, 2006] T. Aoto. Dealing with non-orientable equations in rewriting induction. In Frank Pfenning, editor, *RTA*, volume 4098 of *Lecture Notes in Computer Science*, pages 242–256. Springer, 2006.

[Armando *et al.*, 2002] A. Armando, M. Rusinowitch, and S. Stratulat. Incorporating decision procedures in implicit induction. *Journal of Symbolic Computation*, 34(4):241–258, 2002.

[Aubin, 1979] R. Aubin. Mechanizing structural induction. *Theor. Comput. Sci.*, 9:329–362, 1979.

[Avenhaus *et al.*, 2003] J. Avenhaus, U. Kühler, T. Schmidt-Samoa, and C.-P. Wirth. How to prove inductive theorems? QuodLibet! In Franz Baader, editor, *Proceedings of the 19th International Conference on Automated Deduction (CADE-19)*, number 2741 in Lecture Notes in Artificial Intelligence, pages 328–333. Springer, 2003.

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

[Bachmair and Ganzinger, 2001] L. Bachmair and H. Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 19–99. Elsevier and MIT Press, 2001.

[Bachmair, 1988] L. Bachmair. Proof by consistency in equational theories. *Logic in Computer Science, 1988. LICS '88., Proceedings of the Third Annual Symposium on*, pages 228–233, Jul 1988.

[Barthe and Courtieu, 2002] G. Barthe and P. Courtieu. Efficient reasoning about executable specifications in Coq. In *Theorem Proving in Higher Order Logics*, volume 2410 of *LNCS*, pages 31–46. Springer Berlin, 2002.

[Barthe and Dufay, 2003] G. Barthe and G. Dufay. Verified bytecode verifiers revisited. Manuscript, 2003.

[Barthe and Stratulat, 2003] G. Barthe and S. Stratulat. Validation of the JavaCard platform with implicit induction techniques. In R. Nieuwenhuis, editor, *RTA (Rewriting Techniques and Applications)*, volume 2706 of *LNCS*, pages 337–351. Springer, 2003.

[Barthe *et al.*, 2001a] G. Barthe, G. Dufay, M. Huisman, and S. Melo de Sousa. Jakarta: A toolset for reasoning about javacard. In I. Attali and T. P. Jensen, editors, *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings*, volume 2140 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2001.

[Barthe *et al.*, 2001b] G. Barthe, G. Dufay, L. Jakubiec, B. P. Serpette, and S. Melo de Sousa. A formal executable semantics of the javacard platform. In D. Sands, editor, *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2028 of *Lecture Notes Computer Science*, pages 302–319. Springer, 2001.

[Barthe *et al.*, 2002a] G. Barthe, P. Courtieu, G. Dufay, and S. Melo de Sousa. Tool-assisted specification and verification of the javacard platform. In H. Kirchner and C. Ringessein, editors, *Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, September 9-13, 2002, Proceedings*, volume 2422 of *Lecture Notes in Computer Science*, pages 41–59. Springer, 2002.

[Barthe *et al.*, 2002b] G. Barthe, G. Dufay, L. Jakubiec, and S. Melo de Sousa. A formal correspondence between offensive and defensive javacard virtual machines. In A. Cortesi, editor, *Verification, Model Checking, and Abstract Interpretation, Third International Workshop, VMCAI 2002, Venice, Italy, January 21-22, 2002, Revised Papers*, volume 2294 of *Lecture Notes Computer Science*, pages 32–45. Springer, 2002.

[Basin *et al.*, 2004] D. Basin, Y. Deville, P. Flener, A. Hamfelt, and J. F. Nilsson. Synthesis of programs in computational logic. In *Program Development in Computational Logic*, pages 30–65. Springer, 2004.

[Berardi and Tatsuta, 2019] S. Berardi and M. Tatsuta. Classical system of Martin-Lof's inductive definitions is not equivalent to cyclic proofs. *Logical Methods in Computer Science*, 15(3), 2019.

[Berregeb *et al.*, 1996] N. Berregeb, A. Bouhoula, and M. Rusinowitch. SPIKE-AC: A system for proofs by induction in associative-commutative theories. In H. Ganzinger, editor, *Rewriting Techniques and Applications*, volume 1103 of *Lecture Notes in Computer Science*, pages 428–431. Springer Berlin Heidelberg, 1996.

[Berregeb *et al.*, 1998] N. Berregeb, A. Bouhoula, and M. Rusinowitch. Observational proofs with critical contexts. In *Fundamental Approaches to Software Engineering*, pages 38–53, 1998.

[Bertot and Castéran, 2004] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

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

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

[Bouhoula and Rusinowitch, 1993] A. Bouhoula and M. Rusinowitch. Automatic case analysis in proof by induction. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI)*, volume 1, pages 88–94, 1993.

[Bouhoula and Rusinowitch, 1995] A. Bouhoula and M. Rusinowitch. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 14(2):189–235, 1995.

[Bouhoula and Rusinowitch, 2002] A. Bouhoula and M. Rusinowitch. Observational proofs by rewriting. *Theoretical Computer Science*, 275(1–2):675–698, 2002.

[Bouhoula *et al.*, 1992] A. Bouhoula, E. Kounalis, and M. Rusinowitch. SPIKE, an automatic theorem prover. In *Logic Programming and Automated Reasoning (LPAR)*, pages 460–462, 1992.

[Bouhoula *et al.*, 1995] A. Bouhoula, E. Kounalis, and M. Rusinowitch. Automated mathematical induction. *Journal of Logic and Computation*, 5(5):631–668, 1995.

[Bouhoula, 1994a] A. Bouhoula. *Preuves automatiques par récurrence dans les théories conditionnelles.* PhD thesis, Université Henri Poincaré - Nancy I, 1994.

[Bouhoula, 1994b] A. Bouhoula. SPIKE: A system for sufficient completeness and parameterized inductive proofs. In A. Bundy, editor, *Automated Deduction — CADE-12*, volume 814 of *Lecture Notes in Computer Science*, pages 836–840. Springer Berlin Heidelberg, 1994.

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

[Bouhoula, 1997] A. Bouhoula. Automated theorem proving by test set induction. *Journal of Symbolic Computation*, 23(1):47–77, 1997.

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

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

[Boyer and Moore, 1979] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, NY, 1979.

[Boyer and Moore, 1988a] R. S. Boyer and J S. Moore. *A computational logic handbook.* Academic Press Professional, 1988.

[Boyer and Moore, 1988b] R. S. Boyer and J S. Moore. Integrating decision procedures into heuristic theorem provers: a case study of linear arithmetic. In *Machine Intelligence*, pages 83–124. Oxford University Press, Inc. New York, NY, USA, 1988.

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

[Bronsard and Reddy, 1991] F. Bronsard and U. S. Reddy. Conditional rewriting in Focus. In *Conditional and Typed Rewriting Systems*, pages 1–13, 1991.

[Bronsard *et al.*, 1994] F. Bronsard, U.S. Reddy, and R. Hasker. Induction using term orderings. In *CADE (Conf. on Automated Deduction)*, volume 814 of *LNCS*, pages 102–117. Springer, 1994.

[Bronsard *et al.*, 1996] F. Bronsard, S. R. Uday, and R. W. Hasker. Induction using term orders. *Journal of Automated Reasoning*, 16(1-2):3–37, 1996.

[Brotherston and Simpson, 2011] J. Brotherston and A. Simpson. Sequent calculi for induction and infinite descent. *Journal of Logic and Computation*, 21(6):1177–1216, 2011.

[Brotherston *et al.*, 2012] J. Brotherston, N. Gorogiannis, and R. L. Petersen. A generic cyclic theorem prover. In *APLAS-10 (10th Asian Symposium on Programming Languages and Systems)*, volume 7705 of *LNCS*, pages 350–367. Springer, 2012.

[Brotherston, 2005] J. Brotherston. Cyclic proofs for first-order logic with inductive definitions. In *Proceedings of TABLEAUX-14*, volume 3702 of *LNAI*, pages 78–92. Springer-Verlag, 2005.

[Brotherston, 2006] J. Brotherston. *Sequent Calculus Proof Systems for Inductive Definitions*. PhD thesis, University of Edinburgh, November 2006.

[Buchberger *et al.*, 2016] Bruno Buchberger, Tudor Jebelean, Temur Kutsia, Alexander Maletzky, and Wolfgang Windsteiger. Theorema 2.0: Computer-Assisted Natural-Style Mathematics. *JFR*, 9(1):149–185, 2016.

[Buchberger, 2000] B. Buchberger. Theory Exploration with Theorema. In *Analele Universitatii Din Timisoara, Ser. Matematica-Informatica*, volume XXXVIII, pages 9–32, 2000.

[Bundy *et al.*, 2006] A. Bundy, L. Dixon, J. Gow, and J. Fleuriot. Constructing induction rules for deductive synthesis proofs. *Electron. Notes Theor. Comput. Sci.*, 153:3–21, March 2006.

[Burstall, 1969] R. M. Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12:41–48, 1969.

[Casset *et al.*, 2002] L. Casset, L. Burdy, and A. Requet. Formal development of an embedded verifier for java card byte code. In *2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings*, pages 51–58. IEEE Computer Society, 2002.

[Cervesato *et al.*, 1999] I. Cervesato, N. A. Durgin, P. Lincoln, J. C. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop, CSFW 1999, Mordano, Italy, June 28-30, 1999*, pages 55–69. IEEE Computer Society, 1999.

[Coglio *et al.*, 1998] A. Coglio, A. Goldberg, and Z. Qian. Towards a provably-correct implementation of the JVM bytecode verifier. Formal Underpinnings of Java Workshop at OOPSLA, 1998.

[Colton, 2012] Simon Colton. *Automated Theory Formation in Pure Mathematics*. Springer Science & Business Media, 2012.

[Comon and Nieuwenhuis, 2000] H. Comon and R. Nieuwenhuis. Induction= I-axiomatization+ first-order consistency. *Information and computation(Print)*, 159(1-2):151–186, 2000.

[Comon, 2001] H. Comon. Inductionless induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 913–962. Elsevier and MIT Press, 2001.

[Contejean *et al.*, 2007] E. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Certification of automated termination proofs. *Frontiers of Combining Systems*, pages 148–162, 2007.

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

[Courant, 1996] J. Courant. Proof reconstruction. Research Report RR96-26, LIP, 1996. Preliminary version.

[Cousineau and Mauny, 1998] G. Cousineau and M. Mauny. *The Functional Approach to Programming with Caml*. Cambridge University Press, 1998.

[De Moura and Bjørner, 2008] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[Denker *et al.*, 2000] G. Denker, J. Meseguer, and C. Talcott. Formal specification and analysis of active networks and communication protocols: The Maude experience. DARPA Information Survivability Conference and Exposition (DISCEX 2000). IEEE., 2000.

[Dershowitz and Moser, 2007] Nachum Dershowitz and Georg Moser. The Hydra battle revisited. *Rewriting, Computation and Proof*, pages 1–27, 2007.

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

[Dershowitz, 1982a] N. Dershowitz. Applications of the Knuth-Bendix completion procedure. In *Seminaire d'Informatique Theorique*, pages 95–111, 1982.

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

[Dramnesc and Jebelean, 2011] I. Dramnesc and T. Jebelean. Proof techniques for synthesis of sorting algorithms. In *SYNASC 2011: Proceedings of the 13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 101–109. IEEE Computer Society, 2011.

[Dramnesc *et al.*, 2015a] I. Dramnesc, T. Jebelean, and S. Stratulat. Combinatorial techniques for proof-based synthesis of sorting algorithms. In *SYNASC 2015: Proceedings of the 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 137–144. IEEE Computer Society, 2015.

[Dramnesc *et al.*, 2015b] I. Dramnesc, T. Jebelean, and S. Stratulat. Synthesis of some algorithms for trees: Experiments in Theorema. Technical Report 15-04, Johannes Kepler University, Linz, Austria, 2015.

[Dramnesc *et al.*, 2015c] I. Dramnesc, T. Jebelean, and S. Stratulat. Theory exploration of binary trees. In *13th IEEE International Symposium on Intelligent Systems and Informatics (SISY 2015)*, pages 139–144. IEEE Publishing, 2015.

[Dramnesc *et al.*, 2016a] I. Dramnesc, T. Jebelean, and S. Stratulat. A case study on algorithm discovery from proofs: The insert function on binary trees. In *SACI 2016: 11th IEEE International Symposium on Applied Computational Intelligence and Informatics*, pages 231–236. IEEE, 2016.

[Dramnesc *et al.*, 2016b] I. Dramnesc, T. Jebelean, and S. Stratulat. Proof-based synthesis of sorting algorithms for trees. In *LATA 2016: 10th International Conference on Language and Automata Theory and Applications*, volume 9618 of *Lecture Notes Computer Science*, pages 562–575. Springer Verlag, 2016.

[Dramnesc *et al.*, 2019] I. Dramnesc, T. Jebelean, and S. Stratulat. Mechanical synthesis of sorting algorithms for binary trees by logic and combinatorial techniques. *Journal of Symbolic Computation*, 90:3–41, 2019.

[Fribourg, 1986] L. Fribourg. A strong restriction of the inductive completion procedure. In *ICALP (International Conference on Automata, Languages and Programming)*, volume 226 of *Lecture Notes in Computer Science*, pages 105–115, 1986. (Extended version in Journal of Symbolic Computation, Volume 8, Issue 3, September 1989, Pages 253-276).

[Garland and Guttag, 1988] S. J. Garland and J. V. Guttag. Inductive methods for reasoning about abstract data types. *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 219–228, 1988.

[Gentzen, 1935] G. Gentzen. Untersuchungen über das logische Schließen. I. *Mathematische Zeitschrift*, 39:176–210, 1935.

[Goguen, 1980] J. Goguen. How to prove algebraic inductive hypotheses without induction. In *5th Conference on Automated Deduction (CADE05)*, volume 87 of *Lecture Notes in Computer Science*, pages 356–373. Springer, 1980.

[Gramlich, 2005] B. Gramlich. Strategic issues, problems and challenges in inductive theorem proving. *Electronic Notes in Theoretical Computer Science*, 125(2):5–43, March 2005.

[Gulwani, 2010] S. Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, pages 13–24, New York, NY, USA, 2010. ACM.

[Hartel and Moreau, 2001] P. H. Hartel and L. Moreau. Formalizing the safety of Java, the Java virtual machine, and Javacard. *ACM Comput. Surv.*, 33(4):517–558, 2001.

[Henaien and Stratulat, 2013] A. Henaien and S. Stratulat. Performing implicit induction reasoning with certifying proof environments. In A. Bouhoula, T. Ida, and F. Kamareddine, editors, *Proceedings Fourth International Symposium on Symbolic Computation in Software Science, Gammarth, Tunisia, 15-17 December 2012*, volume 122 of *Electronic Proceedings in Theoretical Computer Science*, pages 97–108. Open Publishing Association, 2013.

[Howe, 1993] D. J. Howe. Reasoning about functional programs in Nuprl. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, pages 145–164. Springer Verlag, 1993.

[Huet and Hullot, 1980] G. Huet and J.M. Hullot. Proofs by induction in equational theories with constructors. Technical Report 0028, INRIA, 1980.

[Huet, 1991] G. Huet. The Gilbreath trick: A case study in axiomatisation and proof development in the Coq proof assistant. Technical Report RR-1511, INRIA, 1991.

[Johansson *et al.*, 2011] M. Johansson, L. Dixon, and A. Bundy. Conjecture synthesis for inductive theories. *Journal of Automated Reasoning*, 47(3):251–289, 2011.

[Jouannaud and Kounalis, 1986] J.-P. Jouannaud and E. Kounalis. Automatic proofs by induction in equational theories without constructors. In A. Meyer, editor, *Proceedings of the First Annual IEEE Symp. on Logic in Computer Science, LICS 1986*, pages 358–366. IEEE Computer Society Press, June 1986.

[Jouannaud and Kounalis, 1989] J.-P. Jouannaud and E. Kounalis. Automatic proofs by induction in theories without constructors. *Information and Computation*, 82(1):1 – 33, 1989.

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

[Kapur and Subramaniam, 1996] D. Kapur and M. Subramaniam. Automating induction over mutually recursive functions. In *Algebraic Methodology and Software Technology*, volume 1101 of *LNCS*, pages 117–131. Springer, 1996.

[Kapur and Zhang, 1988] D. Kapur and H. Zhang. RRL: A rewrite rule laboratory. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction*, volume 310 of *LNCS*, pages 768–769. Springer Berlin / Heidelberg, 1988.

[Kapur and Zhang, 1994] D. Kapur and H. Zhang. Automating induction: Explicit vs. inductionless. *Proc. Third International Symposium on Artificial Intelligence and Mathematics, Fort Lauderdale, Florida, Jan*, pages 2–5, 1994.

[Kapur *et al.*, 1986] D. Kapur, P. Narendran, and H. Zhang. Proof by induction using test sets. In *8th International Conference on Automated Deduction*, volume 230 of *Lecture Notes Computer Science*, pages 99–117. Springer, 1986.

[Kapur *et al.*, 1991] Deepak Kapur, Paliath Narendran, and Hantao Zhang. Automating inductionless induction using test sets. *Journal of Symbolic Computation*, 11(1/2):81–111, 1991.

[Kaufmann and Moore, 1999] M. Kaufmann and J S. Moore. *ACL2 Version 8.3 - The User's Manual*, 1999.

[Kleene, 1936] S. C. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112:727–742, 1936.

[Klein and Nipkow, 2003] G. Klein and T. Nipkow. Verified bytecode verifiers. *Theor. Comput. Sci.*, 298(3):583–626, 2003.

[Klein, 2003] G. Klein. *Verified Java Bytecode Verification.* PhD thesis, Technical University Munich, 2003.

[Knuth and Bendix, 1970] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, pages 263–297, 1970.

[Knuth, 1998] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison Wesley Longman Publishing, Redwood City, CA, USA, 2 edition, 1998.

[Kounalis and Rusinowitch, 1990a] E. Kounalis and M. Rusinowitch. A mechanization of conditional reasoning. In *First International Symposium on Artificial Intelligence and Mathematics*, 1990.

[Kounalis and Rusinowitch, 1990b] E. Kounalis and M. Rusinowitch. Mechanizing inductive reasoning. In *Proceedings of the eighth National conference on Artificial intelligence - Volume 1*, AAAI'90, pages 240–245. AAAI Press, 1990.

[Küchlin, 1989] W. Küchlin. Inductive completion by ground proof transformation. In H. Ait-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures (Volume II): Rewriting Techniques*, pages 211–244. Academic Press, London, 1989.

[Kupferman and Vardi, 2001] O. Kupferman and M. Vardi. Weak alternating automata are not that weak. *ACM Transactions on Computational Logic (TOCL)*, 2(3):408–429, 2001.

[Lankford, 1980] D. Lankford. Some remarks on inductionless induction. Technical Report MTP-11, Math. Dept., Louisiana Tech. Univ., Ruston, 1980.

[Leroy *et al.*,] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system - release 4.00. Documentation and user's manual.* INRIA.

[Lescanne, 1983] P. Lescanne. 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, pages 99–108, New York, NY, USA, 1983. ACM.

[Lescanne, 1990] P. Lescanne. Implementation of completion by transition rules + control: ORME. In H. Kirchner and W. Wechler, editors, *Algebraic and Logic Programming*, volume 463 of *Lecture Notes in Computer Science*, pages 262–269. Springer Berlin Heidelberg, 1990.

[McCarthy and Painter, 1967] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Mathematical Aspects of Computer Science*, pages 33–41. American Mathematical Society, 1967.

[McCarthy, 1963] J. McCarthy. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, pages 33–70. North-Holland, 1963.

[McCasland and Bundy, 2006] R. L. McCasland and A. Bundy. Mathsaid: A mathematical theorem discovery tool. In Viorel Negru, Dana Petcu, Daniela Zaharie, Ajith Abraham, Bruno Buchberger, Alexandru Cicortas, Dorian Gorgan, and Joël Quinqueton, editors, *8th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2006), 26-29 September 2006, Timisoara, Romania*, pages 17–22. IEEE Computer Society, 2006.

[McCune, 1994] W. McCune. OTTER 3.0 reference manual and guide. Technical report, Argonne Nationale Institute, 1994.

[Michel, 1988] M. Michel. Complementation is more difficult with automata on infinite words. Technical report, CNET, 1988.

[Musser, 1980] D. R. Musser. On proving inductive properties of abstract data types. In *POPL*, pages 154–162, 1980.

[Naidich, 1996] D. Naidich. On generic representation of implicit induction procedures. Technical Report CS-R9620, CWI, 1996.

[Negri and v. Plato, 2001] S. Negri and J. v. Plato. *Structural Proof Theory.* Cambridge University Press, 2001.

[Nguyen *et al.*, 2002] Q. H. Nguyen, C. Kirchner, and H. Kirchner. External rewriting for skeptical proof assistants. *J. Autom. Reasoning*, 29(3-4):309–336, 2002.

[Nieuwenhuis and Rubio, 2001] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001.

[Nipkow *et al.*, 2002] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science.* Springer, 2002.

[Nipkow, 2001] T. Nipkow. Verified bytecode verifiers. In F. Honsell and M. Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2030 of *Lecture Notes Computer Science*, pages 347–363. Springer, 2001.

[Paulin-Mohring, 2015] C. Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In B. Woltzenlogel Paleo and D. Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, January 2015.

[Poincaré, 1902] H. Poincaré. *La Science et l'Hypothèse.* Flammarion, 1902.

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

[Rabadan and Klay, 1997] C. Rabadan and F. Klay. Un nouvel algorithme de contrôle de conformité pour la capacité de transfert 'Available Bit Rate'. Technical Report NT/CNET/5476, CNET, 1997.

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

[Rusinowitch *et al.*, 2003] M. Rusinowitch, S. Stratulat, and F. Klay. Mechanical verification of an ideal incremental ABR conformance algorithm. *Journal of Automated Reasoning*, 30(2):153–177, 2003.

[Sprenger and Dam, 2003] C. Sprenger and M. Dam. On the structure of inductive reasoning: Circular and tree-shaped proofs in the μ calculus. In A. Gordon, editor, *Foundations of Software Science and Computation Structures*, volume 2620 of *Lecture Notes in Computer Science*, pages 425–440. Springer Berlin / Heidelberg, 2003.

[Stärk *et al.*, 2001] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation.* Springer Verlag, 2001.

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

[Stratulat, 2000a] S. Stratulat. A general framework to build multi-logic implicit induction provers. In *First International Workshop Freiburg-Genova (FreGe'2000)*, 2000.

[Stratulat, 2000b] S. Stratulat. *Preuves par récurrence avec ensembles couvrants contextuels. Applications à la vérification de logiciels de télécommunications.* PhD thesis, Université Henri Poincaré, Nancy I, November 2000.

[Stratulat, 2001] S. Stratulat. A general framework to build contextual cover set induction provers. *Journal of Symbolic Computation*, 32(4):403–445, 2001.

[Stratulat, 2005] S. Stratulat. Automatic 'Descente Infinie' induction reasoning. In B. Beckert, editor, *TABLEAUX*, volume 3702 of *Lecture Notes in Artificial Intelligence*, pages 262–276. Springer, 2005.

[Stratulat, 2007] S. Stratulat. 'Descente Infinie' induction-based saturation procedures. In *SYNASC '07: Proceedings of the Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 17–24, Washington, DC, USA, 2007. IEEE Computer Society.

[Stratulat, 2008] S. Stratulat. Combining rewriting with Noetherian induction to reason on non-orientable equalities. In A. Voronkov, editor, *Rewriting Techniques and Applications*, volume 5117 of *Lecture Notes in Computer Science*, pages 351–365. Springer Berlin, 2008.

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

[Stratulat, 2012] S. Stratulat. A unified view of induction reasoning for first-order logic. In A. Voronkov, editor, *Turing-100 (The Alan Turing Centenary Conference)*, volume 10 of *EPiC Series*, pages 326–352. EasyChair, 2012.

[Stratulat, 2014] S. Stratulat. Implementing reasoning modules in implicit induction theorem provers. In *SYNASC (International Symposium on Symbolic and Numeric Algorithms for Scientific Computing)*, pages 133–140. IEEE Computer Society, 2014.

[Stratulat, 2016] S. Stratulat. Structural vs. cyclic induction: a report on some experiments with Coq. In *SYNASC (International Symposium on Symbolic and Numeric Algorithms for Scientific Computing)*, pages 27–34. IEEE Computer Society, 2016.

[Stratulat, 2017a] S. Stratulat. Cyclic proofs with ordering constraints. In R. A. Schmidt and C. Nalon, editors, *TABLEAUX 2017 (26th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods)*, volume 10501 of *LNAI*, pages 311–327. Springer, 2017.

[Stratulat, 2017b] S. Stratulat. Mechanically certifying formula-based Noetherian induction reasoning. *Journal of Symbolic Computation*, 80, Part 1:209–249, 2017.

[Stratulat, 2018] S. Stratulat. Validating back-links of $\mathrm{FOL}_{\mathrm{ID}}$ cyclic pre-proofs. In S. Berardi and S. van Bakel, editors, *CL&C'18 (Seventh International Workshop on Classical Logic and Computation)*, number 281 in EPTCS, pages 39–53, 2018.

[Stratulat, 2020] S. Stratulat. SPIKE, an automatic theorem prover – revisited. In *SYNASC 2020: Proceedings of the 22nd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 93–96. IEEE Computer Society, 2020.

[Syme and Gordon, 2002] D. Syme and A. D. Gordon. Automating type soundness proofs via decision procedures and guided reductions. In M. Baaz and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 9th International Conference, LPAR 2002, Tbilisi, Georgia, October 14-18, 2002, Proceedings*, volume 2514 of *Lecture Notes Computer Science*, pages 418–434. Springer, 2002.

[Tarjan, 1972] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146—160, 1972.

[The Coq development team, 2020] The Coq development team. *The Coq Reference Manual*. INRIA, 2020. http://coq.inria.fr/doc.

[The SPIKE development team, 2020] The SPIKE development team. *The SPIKE prover*, 2020. https://github.com/sorinica/spike-prover.

[Turing, 1936] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[Walther, 1994] C. Walther. Mathematical induction. In Dov M. Gabbay, Christopher J. Hogger, J. A. Robinson, and Jörg H. Siekmann, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming (2)*, pages 127–228. Oxford University Press, 1994.

[Wirth and Gramlich, 1994] C.-P. Wirth and B. Gramlich. On notions of inductive validity for first-order equational clauses. *Automated Deduction —CADE-12*, pages 162–176, 1994.

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

[Wirth, 2005] C.-P. Wirth. History and future of implicit and inductionless induction: Beware the old jade and the zombie ! In *Mechanizing Mathematical Reasoning: Essays in Honor of Jörg H. Siekmann on the Occasion of His 60th Birthday*, number 2605 in Lecture Notes in Artificial Intelligence, pages 192–203. Springer, 2005.

[Wolfram, 2003] S. Wolfram. *The Mathematica Book*. Wolfram Media Inc., 2003.

[Zhang *et al.*, 1988] H. Zhang, D. Kapur, and M. S. Krishnamoorthy. A mechanizable induction principle for equational specifications. In *Proceedings of the 9th International Conference on Automated Deduction*, pages 162–181, London, UK, 1988. Springer-Verlag.

Index

E-Cyclist, 101
'Descente Infinie' induction, ix

algorithm synthesis, 36

bud, 91

companion, 91
compatible induction orderings, 20
completeness property, 2
conjecture, 7
contextual cover set, 9
counterexample, 2
cover set, 5
cover set induction, 5
cover substitutions, 6
cumulative substitution, 96

deductive consequence, 2
deductive relation, 2
DRaCuLa strategy, 13

eager induction, 5
enhanced rewriting induction, 9

fairness of an inference system, 48
formula cover sets, 6
formula-based Noetherian induction, 4

global trace condition, 93
ground confluence, 6

hierarchical induction, 9

IH-node, 94
implicit induction, 6
incremental rewriting induction, 9
induction function, 91
induction hypothesis, ix
induction hypothesis discharged by a SCC, 99
induction schema, 5
induction variable, 5
inductionless induction, 6
inductive antecedent atom, 92
inductive consequence, 2
inductive predicate, 90
inductive relation, 2
inductive theory, 2
inference rule, 7
inference system, x
infinitely progressing trace, 93
initial model, 2
interpretation, 2

lazy induction, 10
leaf, 91
lexicographic extension of an ordering, 3
logical consequence, 2

model, 2
most general unifier, 9
multiset extension of an ordering, 3
multiset path ordering, 3
mutual induction, 10

Noetherian induction, ix

ordered rewriting induction, 9
ordering-derivability, 97
overall identity substitution, 96

Peano induction principle, viii
pre-proof tree, 91
pre-proof tree-sets, 93
premise, 7
progress point, 93
proof, 7
proof by consistency, 6
proof derivation, 7

rb-path, 94
recursive inference system, 110
recursive path ordering, 3
reduction ordering, 3
reductive inference system, 7
redundant formula w.r.t. other formulas, 117
refutational completeness, 6
relation stable under contexts, 3
relation stable under substitutions, 3
relaxed rewriting, 9

rewrite operation, 69

sequent, 90
soundness property, 2
status of a function symbol, 3
strict cover set, 6
strongly connected component, 28
structural induction, 5
subsumption, 117
sufficient completeness, 2
synthesis conjecture, 36

term-based Noetherian induction, 4
term-rewriting induction, 6
test sets, 6
the inference system A, 46
the inference system D, 14
the inference system D', 74
the inference system D_c, 19
the inference system I_s^f, 70
the inference system I_c^b, 74
the inference system I_c^f, 80
the inference system I_i, 8
the inference system I_s, 8
the inference system G, 113
the inference system G', 114
the inference system A', 113
the inference system A^1, 109
the inference system A^2, 109
the inference system A_s, 115
the inference system P, 112
the inference system RP, 117
the inference system RP', 117
the inference system I, 7
the inference system I_s^b, 67
the inference system CLKID_N^ω, 90
the inference system LKID_a, 124
total quasi-order, 3
trace, 92

validity, 2

well-founded induction principle, ix

Glossary

BCV: ByteCode Verifier
CCS: Contextual Cover Set
FOL: First-Order Logic
$\mathbf{FOL_{ID}}$: First-Order Logic with Inductive Definitions
IAA: Inductive Antecedent Atom
IC: Induction Case
IH: Induction Hypothesis
JCVM: JavaCard Virtual Machine
JSL: Jakarta Specification Language
JTL: Jakarta Transformation Kit
mgu: most general unifier
mpo: multiset path ordering
rpo: recursive path ordering
SCC: Strongly Connected Component
VM: Virtual Machine

Résumé

Le principe de la récurrence noethérienne est un des plus généraux principes du raisonnement formel. Dans le cadre du raisonnement de premier ordre, nous proposons une classification de ses instances pouvant être partagées en instances basées sur des termes et des formules. Nous donnons un aperçu du raisonnement par récurrence noethérienne basé sur des termes et des formules, et établissons des relations entre eux. Nous montrons que toute preuve intégrant du raisonnement par récurrence noethérienne basée sur des termes peut être convertie en une preuve dont le raisonnement par récurrence est basé sur des formules. La question de la conversion dans l'autre direction reste ouverte. Pourtant, nous identifions certaines classes de preuves par récurrence noethérienne basée sur des formules qui peuvent être traduites en des preuves dont le raisonnement par récurrence est basé sur des termes. Nous établissons des liens entre le raisonnement noéthérien basé sur des formules et d'autres types de raisonnement formel de premier ordre, comme le raisonnement par récurrence cyclique pour la logique de premier ordre avec des définitions inductives ($\mathrm{FOL_{ID}}$) et le raisonnement basé sur la saturation. Nous avons mis au point des méthodologies pour certifier le raisonnement noethérien basé sur des formules et le raisonnement cyclique pour $\mathrm{FOL_{ID}}$ en utilisant l'assistant de preuve Coq.

Les forces et les limites de nos résultats ont été illustrées par des exemples, des études de cas non-triviaux et des expériences informatiques. Les développements logiciels les plus importants sont une nouvelle version du prouveur SPIKE ainsi que E-Cyclist, l'extension du prouveur Cyclist avec une nouvelle méthode de vérification de la correction de ses pré-preuves dans $\mathrm{FOL_{ID}}$.

Mots-clés: raisonnement par récurrence noethérienne, raisonnement formel de premier ordre, certification de preuves, SPIKE, Coq.

Abstract

Noetherian induction is one of the most general induction principles used in formal reasoning. In the frame of the first-order reasoning, we propose a classification of its instances that can be split into term- and formula-based instances. We give an overview of the term- and formula-based Noetherian induction reasoning, and established relations between them. We show that every term-based Noetherian induction proof can be converted to a formula-based one. The question about the conversion in the other direction remains open. However, we identify certain classes of formula-based Noetherian induction proofs that can be translated into term-based ones. We establish connections between formula-based Noetherian reasoning and other kinds of first-order reasoning, as the cyclic induction reasoning for first-order logic with inductive definitions ($\mathrm{FOL_{ID}}$) and saturation-based reasoning. Last but not least, we have devised methodologies for certifying formula-based Noetherian induction proofs and $\mathrm{FOL_{ID}}$ cyclic proofs using the Coq proof assistant.

The strengths and limits of our results have been illustrated by examples, non-trivial case studies and computer experiments. The most important software developments are the new version of the SPIKE prover and E-Cyclist, the extension of the $\mathrm{FOL_{ID}}$ prover Cyclist with a new checking method for the soundness of its pre-proofs.

Keywords: Noetherian induction reasoning, first-order formal reasoning, proof certification, SPIKE, Coq.