HAL
open science

Reasoning on Schemata of Formulæ

Mnacho Echenim, Nicolas Peltier

To cite this version:

Mnacho Echenim, Nicolas Peltier. Reasoning on Schemata of Formulæ. AISC 2012 - Artificial Intelligence and Symbolic Computation (Part of CICM 2012), Jul 2012, Bremen, Germany. pp.310-325, 10.1007/978-3-642-31374-5_21 . hal-00933516

HAL Id: hal-00933516

https://hal.science/hal-00933516v1

Submitted on 20 Jan 2014

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.

Reasoning on Schemata of Formulæ*

Mnacho Echenim and Nicolas Peltier

University of Grenoble** (LIG, Grenoble INP/CNRS)

Abstract. A logic is presented for reasoning on iterated sequences of formulæ over some given base language. The considered sequences, or *schemata*, are defined inductively, on some algebraic structure (for instance the natural numbers, the lists, the trees etc.). A proof procedure is proposed to relate the satisfiability problem for schemata to that of finite disjunctions of base formulæ. It is shown that this procedure is sound, complete and terminating, hence the basic computational properties of the base language can be carried over to schemata.

1 Introduction

We introduce a logic for reasoning on iterated schemata of formulæ. The schemata we consider are infinite sequences of formulæ over a given *base language*, and these sequences are defined by induction on some algebraic structure (e.g. the natural numbers). As an example, consider the following sequence of propositional formulæ ϕ_n, parameterized by a natural number n:

$$\phi_0 \rightarrow \top \qquad \phi_{n+1} \rightarrow \phi_n \wedge (p(n) \Leftrightarrow p(n+1)).$$

It is clear that the formula $\phi_n \wedge p(0) \wedge \neg p(n)$ is unsatisfiable, for every $n \in \mathbb{N}$. This can be easily checked by any SAT-solver, for every *fixed* value of n. Here the base language is propositional logic and the sequence is defined over the natural numbers. However, proving that it is is unsatisfiable *for every* $n \in \mathbb{N}$ is a much harder task which obviously requires the use of mathematical induction. Similarly, consider the sequence:

$$\psi_{nil} \rightarrow \top \qquad \psi_{cons(x,y)} \rightarrow \psi_y \wedge (\exists u\, p(y,u)) \Leftrightarrow (\exists v\, p(cons(x,y),v))$$

Then $\psi_l \wedge p(nil,a) \wedge \forall u\, \neg p(l,u)$ is unsatisfiable, for every (finite) list l. Here the base language is first-order logic and the sequence is defined over the set of lists. Such inductively defined sequences are ubiquitous in mathematics and computer science. They are often introduced to analyze the complexity of proof procedures. From a more practical point of view, schemata of propositional formulæ are used to model properties of circuits parameterized by natural numbers, which can represent, e.g., the number of bits, number of layers etc. (see for instance [15], where

* This work has been partly funded by the project ASAP of the French *Agence Nationale de la Recherche* (ANR-09-BLAN-0407-01).

** emails: `Mnacho.Echenim@imag.fr`, `Nicolas.Peltier@imag.fr`

a language is introduced to denote inductively defined boolean functions which can be used to model such parameterized circuits). In mathematics, schemata of first-order formulæ can model inductive proofs, which can be seen as infinite (unbounded) sequences of first-order formulæ (see [5] for an example of the use of this technique in proof analysis).

We now provide a slightly more complex example. The following schema ψ_t encodes a multiplexer, inductively defined as follows. The base case is denoted by $Base(x)$, where x denotes an arbitrary signal. In this case, the output of the circuit is simply the output of x, denoted by $signal(x)$. The inductive case is denoted by $Ind(i, x, y)$, where i is a select input and x and y are two smaller instances of the multiplexer. Its output is either the output of x or that of y, depending on the value of i.

$$\begin{array}{ll} \psi_{Base(x)} & \rightarrow out(Base(x)) \Leftrightarrow signal(x) \\ \psi_{Ind(i,x,y)} & \rightarrow \quad (\neg signal(i) \vee (out(Ind(x,y)) \Leftrightarrow out(x))) \\ & \quad \wedge (signal(i) \vee (out(Ind(x,y)) \Leftrightarrow out(y))) \\ & \quad \wedge \psi_x \wedge \psi_y \end{array}$$

Note that this kind of circuit cannot be encoded in the language of (regular) propositional schemata defined in [2, 3], because the number of inputs is exponential in the depth of the circuit. Hence, the use of non-monadic function symbols is mandatory.

In this paper, we devise a proof procedure to check the satisfiability of these sequences. More precisely, we introduce a formal language for modeling sequences of formulæ defined over an arbitrary base language (encoded as first-order formulæ interpreted in some particular theory) and we show that the computational properties of the base logic carry over to these schemata: If the satisfiability problem is decidable (resp. semi-decidable) for the base language then it is also decidable (resp. semi-decidable) for the corresponding schemata. For instance, the satisfiability problem is decidable for schemata of propositional formulæ and semi-decidable for schemata of first-order formulæ. The basic principle of our proof procedure consists in relating the satisfiability of any iterated schemata of formulæ to that of a *finite* disjunction of base formulæ. The complexity of the satisfiability problem, however, is not preserved in general, since the number of formulæ in the disjunction may be exponential.

This work generalizes previous results [2, 3] in two directions: first the base language is no longer restricted to propositional logic[1] and second the sequences are defined over arbitrary algebraic structures, and not only over the natural numbers. Abstracting from the base language leads to an obvious gain in applicability since our approach now applies to any logic, provided a proof procedure exists for testing the satisfiability of base formulæ. Besides, it has the advantage that the reasoning on schemata is now clearly separated from the reasoning on formulæ in the base language, which may be postponed. This should make our approach much more scalable, since any existing system could now be used as a

[1] A first extension to some decidable theories such as Presburger arithmetic was considered in [4].

"black box" to handle the basic part of the reasoning (whereas the two aspects were closely interleaved in our first approach, yielding additional computational costs). Both extensions significantly increase the scope of our approach.

The extension to arbitrary structures turns out to be the most difficult from a theoretical point of view, mainly because, as we shall see, the number of parameters can increase during the decomposition phase, yielding an increase of the number of related non-decomposable formulæ in each branch, which can in principle prevent termination. In contrast to what happens in the simpler case of propositional schemata [2], these formulæ *cannot* in general be deleted by the purity principle, since they are not independent from the other formulæ in the branch. To overcome this problem, we devise a specific instantiation strategy based on a careful analysis of the depth of terms represented by the parameters, and we define a new loop detection mechanism. This blocking rule is more general and more complex than the one in [2]. We show that it is general enough – together with the proposed instantiation strategy – to ensure termination. Termination is however much more difficult to prove than for propositional schemata defined over natural numbers.

The types of structures that can be handled are quite general: they are defined by sets of – possibly non-free – constructors on a sorted signature. The terms can possibly contain elements of a non-inductive sort. For instance, a list may defined inductively on an *arbitrary* set of elements.

Related Work

There exist many logics and frameworks in which the previous schemata can be encoded, for instance higher-order logic [7]), first-order μ-calculus [18], or logics with inductive definitions [1] that are widely used in proof assistants [19]. However, the satisfiability problem is not even semi-decidable for these logics (due to Gödel's famous result). Very little published research seems to be focused on the identification of complete subclasses and iterated schemata definitely do not lie in these classes and cannot be reduced to them either. Our approach ensures that the basic computational properties of the base language (decidability or semi-decidability) are preserved, at the cost of additional restrictions on the syntax of the schemata under consideration. Furthermore, the modeling of schemata in higher-order languages, although possible from a theoretical point of view, is cumbersome and not very natural in practice.

There exist several approaches in inductive theorem proving, ranging from explicit induction approaches (see for instance [11] or [6]) used mainly by proof assistants to implicit induction schemes used in rewrite-based theorem provers [8, 9], or even to inductionless induction [16, 12], where inductive validity is reduced to a mere satisfiability check. Such approaches can in principle handle some of the formulæ we consider in the present work, provided the base language can be axiomatized. Existing approaches are usually only complete for refutation, in the sense that false conjectures can be disproved, but that inductive theorems cannot always be recognized (this is theoretically unavoidable). Once again, very few termination results exist for such provers and our language

does not fall in the scope of the known complete classes (see for instance [14]). In general, inductive theorem proving requires strong human guidance, especially for specifying the needed inductive lemmata. In contrast, our procedure is *purely automatic*. Of course, this comes at the expense of strongly reducing the form of the inductive axioms. Furthermore, although very restricted to ensure termination and/or completeness, our language allows for more general queries, possibly containing nested quantifiers, which are in general out of the scope of existing automated inductive theorem provers. Indeed, most existing approaches aim at establishing the inductive validity of universal queries w.r.t. a first-order axiomatization (usually a set of clauses). In contrast, our method can handle more general goals of the form $\forall \boldsymbol{x}\, \phi$, where $\boldsymbol{x}$ is a vector of variables interpreted over the considered algebraic structure and ϕ is a formula containing arbitrary quantifiers in the base language.

Practical attempts to use existing inductive theorem provers (such as ACL [10]) to check the satisfiability of schemata such as those in the Introduction fail for every formula except the most trivial ones. We believe that this is not due to a lack of efficiency, but rather to the fact that additional inductive lemmata are required, which cannot be generated automatically by the systems. In some sense, our method (and especially the loop detection rule) can be viewed as an automatic way to generate such lemmata. Our method is also more modular: we make a clear distinction between the reasoning over the base logic and the one over inductive definitions. Inference rules are devised for the latter and an external prover is used to establish the validity of formulæ in the base language.

Since parameterized schemata can obviously be seen as monadic predicates, a seemingly natural idea would be to encode them in monadic second-order logic and use an automata-based approach (see, e.g., [17]) to solve the satisfiability problem. However, as we shall see in Section 3, the unfolding of the inductive definitions contained in a given formula may well increase the number of parameters occurring in it. Since these parameters may share subterms, the formulæ containing them are *not* independent hence they must be handled simultaneously, in the same branch. Thus a systematic decomposition into monadic atoms (in the style of automata-based approaches) is not feasible.

Due to space restrictions, the proofs are omitted and can be found in [13].

2 A Logic for Iterated Schemata

The schemata we consider in this paper are encoded as first-order formulæ, together with a set of rewrite rules specifying the interpretation of certain monadic predicate symbols. Our language is *not* a subclass of first-order logic: indeed, some sort symbols will be interpreted on an inductively defined domain (e.g. on the natural numbers). Furthermore, the formulæ can be interpreted modulo some particular theory, specified by a class of interpretations.

We first briefly review usual notions and notations. We consider first-order terms and formulæ defined on a sorted signature. Let $\mathcal{S}$ be a set of *sort symbols*. Let Σ be a set of *function symbols*, together with a function *profile* mapping

every symbol in Σ to a unique non-empty sequence of elements of $\mathcal{S}$. We write $f : \mathtt{s_1} \times \cdots \times \mathtt{s_n} \to \mathtt{s}$ if *profile*$(f) = \mathtt{s_1}, \ldots, \mathtt{s_n}, \mathtt{s}$ with $n > 0$, and $a : \mathtt{s}$ if *profile*$(a) = \mathtt{s}$ (in this case a is a *constant symbol*). A symbol is *of sort* $\mathtt{s}$ and *of arity* n if its profile is of the form $\mathtt{s_1}, \ldots, \mathtt{s_n}, \mathtt{s}$ (possibly with $n = 0$). The set of function symbols of sort $\mathtt{s}$ is denoted by $\Sigma_\mathtt{s}$. Let $(\mathcal{V}_\mathtt{s})_{\mathtt{s}\in\mathcal{S}}$ be a family of pairwise disjoint set of *variables of sort* $\mathtt{s}$, and $\mathcal{V} \stackrel{\text{def}}{=} \bigcup_{\mathtt{s}\in\mathcal{S}} \mathcal{V}_\mathtt{s}$. We denote by $T_\mathtt{s}$ the sets of *terms of sort* $\mathtt{s}$ built as usual on Σ and $\mathcal{V}$. A term not containing any variable is *ground*.

Definition 1. *Let $\mathcal{I}$ be a subset of $\mathcal{S}$. The elements of $\mathcal{I}$ are called the* inductive sorts. *An $\mathcal{I}$-*term *is a term of a sort $\mathtt{s} \in \mathcal{I}$.*

Let $\mathcal{C} \subseteq \Sigma$ be a set of constructors*, such that the sort of every symbol in $\mathcal{C}$ is in $\bigcup_{\mathtt{s}\in\mathcal{I}} \Sigma_\mathtt{s}$ and such that every non-constant symbol of a sort in $\bigcup_{\mathtt{s}\in\mathcal{I}} \Sigma_\mathtt{s}$ is in $\mathcal{C}$. A* parameter *is a constant symbol of a sort occurring in the profile of a constructor (parameters are denoted by upper-case letters). A term containing only function symbols in $\mathcal{C}$ and variables of sorts in $\mathcal{S} \setminus \mathcal{I}$ is a* constructor term.

Constructors of a sort $\mathtt{s} \in \mathcal{I}$ are meant to define the domain of $\mathtt{s}$, see Definition 5. The constant symbols that are not constructors can be seen as existential variables denoting arbitrary elements of a sort in $\mathcal{I}$ (notice however that $\mathcal{C}$ possibly contains constant symbols). We assume that $\mathcal{I}$ contains a sort symbol $\mathtt{nat}$, with two constructors $0 : \mathtt{nat}$ and $\mathrm{succ} : \mathtt{nat} \to \mathtt{nat}$.

Example 1. Assume that we intend to reason on lists of elements of an arbitrary sort $\mathtt{s}$. Then $\mathcal{S}$ contains the sort symbols $\mathtt{s}$ and $\mathtt{list}$, where $\mathcal{I} = \{\mathtt{list}\}$. The constructors are $nil : \mathtt{list}$ and $cons : \mathtt{s} \times \mathtt{list} \to \mathtt{list}$. The set of parameters contains constant symbols of sorts $\mathtt{s}$ or $\mathtt{list}$ (denoting respectively elements and lists). If A_1, A_2 are parameters of sort $\mathtt{s}$, then $cons(A_1, cons(A_2, nil))$ is a term of sort $\mathtt{list}$.

Similarly, if one wants to reason on lists of natural numbers, then one should take $\mathcal{I} = \mathcal{S} = \{\mathtt{nat}, \mathtt{list}\}$. In this case, $\mathcal{C} = \{nil : \mathtt{list},\ cons : \mathtt{nat} \times \mathtt{list} \to \mathtt{list},\ 0 : \mathtt{nat},\ \mathrm{succ} : \mathtt{nat} \to \mathtt{nat}\}$.

Let $(\mathcal{D}_\mathtt{s})_{\mathtt{s}\in\mathcal{I}}$ be a family of disjoint sets of *defined symbols* of sort $\mathtt{s}$, disjoint from Σ, and $\mathcal{D} \stackrel{\text{def}}{=} \bigcup_{\mathtt{s}\in\mathcal{I}} \mathcal{D}_\mathtt{s}$. An *atom* is either an *equation* of the form $t \simeq s$, where t, s are terms of the same sort, or a *defined atom*, of the form d_t, where $d \in \mathcal{D}_\mathtt{s}$, for some $\mathtt{s} \in \mathcal{I}$, and $t \in T_\mathtt{s}$. The arguments of the symbols in $\mathcal{D}$ are written as indices in order to distinguish them from predicate symbols that may occur in Σ (such predicate symbols may be encoded as functions of profile $\mathtt{s} \to \mathtt{bool}$). Formulæ are built as usual on this set of atoms using the connectives $\vee, \wedge, \neg, \forall, \exists$. We assume for simplicity that all formulæ are in Negation Normal Form (NNF). A variable x is *free* in ϕ if it occurs in ϕ, but not in the scope of the quantifier $\forall x$ or $\exists x$. If ϕ has no free variables then ϕ is *closed*.

An *interpretation* I maps every sort $\mathtt{s}$ to a set of elements $\mathtt{s}^I$, every variable x of sort $\mathtt{s}$ to an element $x^I \in \mathtt{s}^I$, every function symbol $f : \mathtt{s_1} \times \cdots \times \mathtt{s_n} \to \mathtt{s}$ to a function f^I from ${\mathtt{s_1}}^I \times \cdots \times {\mathtt{s_n}}^I$ to $\mathtt{s}^I$ and every defined symbol $d \in D_\mathtt{s}$ to a subset of $\mathtt{s}^I$. The set $\bigcup_{\mathtt{s}\in\mathcal{S}} \mathtt{s}^I$ is the *domain* of I. As usual, any interpretation I can be extended to a function mapping every term t of sort $\mathtt{s}$ to an element

$[t]^I \in \mathtt{s}^I$ and every formula ϕ to a truth value $[\phi]^I \in \{\text{true}, \text{false}\}$. We write $I \models \phi$ (and we say that I *validates* ϕ) if $[\forall \boldsymbol{x}\, \phi]^I = \text{true}$, where $\boldsymbol{x}$ is the vector of free variables in ϕ. We assume, w.l.o.g., that the sets $\mathtt{s}^I$ (for $\mathtt{s} \in \mathcal{S}$) are disjoint. Sets of formulæ are interpreted as conjunctions. If ϕ and ψ are two formulæ or sets of formulæ, we write $\phi \equiv_I \psi$ if either $I \models \phi$ and $I \models \psi$ or $I \not\models \phi$ and $I \not\models \psi$. We write $\phi \equiv \psi$ if $\phi \equiv_I \psi$ for all interpretations I.

We introduce two transformations operating on interpretations. The first one is simple: it only affects the value of some variables or constant symbols. If I is an interpretation, $x_1, \ldots, x_n$ are distinct variables or constant symbols of sort $\mathtt{s_1}, \ldots, \mathtt{s_n}$ respectively and $v_1, \ldots, v_n$ are elements of ${\mathtt{s_1}}^I, \ldots, {\mathtt{s_n}}^I$, then we denote by $I[v_1/x_1, \ldots, v_n/x_n]$ the interpretation coinciding with I, except that for every $i = 1, \ldots, n$, we have: ${x_i}^{I[v_1/x_1,\ldots,v_n/x_n]} \stackrel{\text{def}}{=} v_i$.

The second transformation is slightly more complex. The idea is to change the values of the elements of an inductive sort, without affecting the remaining part of the interpretation. An $\mathcal{I}$-*mapping* for an interpretation I is a function λ mapping every element e in the domain of I to an element of the same sort, that is the identity on every element occurring in a set $\mathtt{s}^I$, where $\mathtt{s} \notin \mathcal{I}$. Then $\lambda(I)$ is the interpretation coinciding with I, except that for every symbol f of a sort $\mathtt{s} \notin \mathcal{I}$, we have: $f^{\lambda(I)}(e_1, \ldots, e_n) \stackrel{\text{def}}{=} f^I(\lambda(e_1), \ldots, \lambda(e_n))$.

In the following, we assume that all interpretations belong to a specific class $\mathfrak{I}$. This is useful to fix the semantics of some of the symbols, for instance one may assume that the interpretation of a sort `int` is not arbitrary but rather equal to $\mathbb{Z}$. Of course, $\mathfrak{I}$ is not arbitrary: the following definitions specify all the conditions that must be satisfied by the considered class of interpretations. We start by the interpretation of the defined symbols. As explained in the Introduction, the value of these symbols are to be specified by convergent systems of rewriting rules, satisfying some additional conditions defined as follows:

Definition 2. *Let $<$ be an ordering on defined symbols. Let $\mathfrak{R}$ be an orthogonal system of rules of the form $d_{f(x_1,\ldots,x_n)} \to \phi$, where d is a defined symbol in $\mathtt{s}$, f is of profile $\mathtt{s_1} \times \cdots \times \mathtt{s_n} \to \mathtt{s}$, and $x_1, \ldots, x_n$ are distinct variables of sorts $\mathtt{s_1}, \ldots, \mathtt{s_n}$. We assume that ϕ and $\mathfrak{R}$ satisfy the following conditions:*

1. *The free variables of ϕ occur in $x_1, \ldots, x_n$.*
2. *All $\mathcal{I}$-terms occurring in ϕ belong to the set $\{x_1, \ldots, x_n, f(x_1, \ldots, x_n)\}$.*
3. *If ϕ contains a formula d'_t then either $d' < d$ and $t = f(x_1, \ldots, x_n)$, or $t \in \{x_1, \ldots, x_n\}$.*
4. *For every constructor f, $\mathfrak{R}$ contains a rule of the form $d_{f(x_1,\ldots,x_n)} \to \phi$.*

It is clear from the conditions of Definition 2 that $\mathfrak{R}$ is convergent (the condition on the ordering ensures termination, and orthogonality ensures confluence). We denote by $d_t{\downarrow_{\mathfrak{R}}}$ the normal form of d_t w.r.t. $\mathfrak{R}$. The following condition states that the interpretation of defined symbols must correspond to the one specified by the rewrite system $\mathfrak{R}$, for every interpretation in $\mathfrak{I}$.

Definition 3. *An interpretation is* $\mathfrak{R}$-compatible *iff for all sort symbols $\mathtt{s} \in \mathcal{I}$, for all defined symbols $d \in \mathcal{D}_{\mathtt{s}}$, for all function symbols $f : \mathtt{s_1} \times \cdots \times \mathtt{s_n} \to \mathtt{s}$, we have $d_{f(x_1,\ldots,x_n)} \equiv_I d_{f(x_1,\ldots,x_n)}{\downarrow_{\mathfrak{R}}}$.*

The second condition that is required ensures that any equation between two constructor terms can be reduced to equations between variables:

Definition 4. *An interpretation is* $\simeq$-decomposable *iff the following conditions hold:*

1. *For every* $\mathtt{s} \in \mathcal{I}$ *and for every* $f, g \in \Sigma_{\mathtt{s}}$ *of arity* n *and* m *respectively, there exists a formula* $\Delta^{(f,g)}$ *built on* $\vee, \wedge, \simeq$ *and on* $n+m$ *distinct variables* $x_1, \ldots, x_n, y_1, \ldots, y_m$ *such that* $f(x_1, \ldots, x_n) \simeq g(y_1, \ldots, y_m) \equiv_I \Delta^{(f,g)}$.
2. *For every* $i \in [1, n]$ *we have* $\Delta^{(f,g)} \models \bigvee_{k=1}^{m} x_i \simeq y_k$, *and for every* $j \in [1, m]$, *we have* $\Delta^{(f,g)} \models \bigvee_{k=1}^{n} y_j \simeq x_k$.

If $t = f(t_1, \ldots, t_n)$ *and* $s = g(s_1, \ldots, s_m)$ *are two non-variable* $\mathcal{I}$*-terms, we denote by* $\Delta(t \simeq s)$ *the formula obtained from* $\Delta^{(f,g)}$ *by replacing each variable* x_i $(1 \leq i \leq n)$ *by* t_i *and each variable* y_j $(1 \leq j \leq m)$ *by* s_j.

Example 2. If, for instance, elements of a sort $\mathtt{s} \in \mathcal{I}$ are interpreted as terms built on a set of free constructors, then we have $\Delta^{(f,g)} \simeq \bot$ if $f \neq g$ and $\Delta^{(f,f)} \stackrel{\text{def}}{=} x_1 \simeq y_1 \wedge \cdots \wedge x_n \simeq y_n$ (where n denotes the arity of f). Indeed, in this case, we have $f(x_1, \ldots, x_n) \simeq f(y_1, \ldots, y_n) \equiv (x_1 \simeq y_1 \wedge \cdots \wedge x_n \simeq y_n)$. If, on the other hand, g is intended to denote a commutative binary function then we should have: $\Delta^{(g,g)} = (x_1 \simeq y_1 \wedge x_2 \simeq y_2) \vee (x_1 \simeq y_2 \wedge x_2 \simeq y_1)$. The variables x_i and y_j are those introduced in Definition 4.

The third condition ensures that the interpretation of every inductive sort is minimal (w.r.t. to set inclusion).

Definition 5. *An interpretation is* $\mathcal{I}$-inductive *iff for every* $\mathtt{s} \in \mathcal{S}$, *and for every element* $u \in \mathtt{s}^I$, *there exists a constructor term* t *such that* $u = [t]^I$.

Notice that, by definition, a constructor term contains no variable of a sort in $\mathcal{I}$. For instance, every element in $\mathtt{nat}^I$ should be equal to a ground term $\mathrm{succ}^k(0)$, for some $k \in \mathbb{N}$. If $\mathtt{list}$ denotes the sort of the lists built on elements of a sort $\mathtt{s} \notin \mathcal{I}$, then any element of $\mathtt{list}^I$ must be equal to a term of the form $cons(x_1, cons(x_2, \ldots, cons(x_n, nil) \ldots))$, where $x_1, \ldots, x_n$ are variables of sort $\mathtt{s}$. This condition implies in particular that for every $\mathtt{s} \notin \mathcal{I}$ and for every element $v \in \mathtt{s}^I$, there exists a variable x such that $x^I = v$ (this is obviously not restrictive, since the variables may be interpreted arbitrarily).

The next definition summarizes all the conditions that are imposed:

Definition 6. *A class of interpretations* $\mathfrak{I}$ *is* schematizable *iff all interpretations* $I \in \mathfrak{I}$ *satisfy the following properties:*

1. I *is* $\mathfrak{R}$*-compatible.*
2. I *is* $\simeq$*-decomposable.*
3. I *is* $\mathcal{I}$*-inductive.*
4. *For all variables* v *of a sort* $\mathtt{s}$ *and for all elements* $e \in \mathtt{s}^I$, $I[e/v] \in \mathfrak{I}$.
5. *For all* $\mathcal{I}$*-mappings* λ, $\lambda(I) \in \mathfrak{I}$.

A formula ϕ *is* $\mathfrak{I}$-satisfiable *iff* ϕ *has a model in* $\mathfrak{I}$.

From now on we focus on testing $\mathfrak{I}$-satisfiability for a schematizable class of interpretations. Before that we impose some restrictions on the formulæ to be tested. As we shall see, these conditions will be useful mainly to ensure that the proof procedure presented in Section 3 only generates a finite number of distinct formulæ, up to a renaming of the parameters. This property is essential for the proof of termination, although it is not a sufficient condition.

Definition 7. *A class of formulæ $\mathfrak{F}$ is* admissible *if all formulæ $\phi \in \mathfrak{F}$ satisfy the following properties:*

1. *For all parameters A, B, $\phi[B/A] \in \mathfrak{F}$.*
2. *ϕ contains no constructor and no variable of a sort in $\mathcal{I}$.*
3. *For every subformula ψ of ϕ, if ψ is not a disjunction, a conjunction, or a defined atom, then ψ contains no defined symbol and no pairs of distinct parameters.*
4. *For every defined symbol d occurring in ϕ and for every rule $d_t \to \phi$ in $\mathfrak{R}$, the formula obtained from ϕ by replacing each $\mathcal{I}$-term by an arbitrary parameter is in $\mathfrak{F}$.*

A formula occurring in $\mathfrak{F}$ is a schema*. It is a* base formula *iff it contains no defined symbol, and no equation between parameters.*

The conditions in Definition 7 ensure that the formulæ in $\mathfrak{F}$ are boolean combinations (built on $\vee$,$\wedge$) of base formulæ containing at most one parameter, of defined atoms and of equations and disequations between parameters. The definition of base formulæ in Definition 7 ensures that the truth values of base formulæ do not depend on the interpretation of the parameters, but only on the *relation* between them. Base formulæ can contain parameters, but they can only occur as arguments of function symbols, whose images must be of a non-inductive sort. The only way of specifying properties of the parameters themselves (and not of the terms built on them) is by using the rewrite rules in $\mathfrak{R}$. As we shall see, this property is essential for proving the soundness of the loop detection rule that ensures termination of our proof procedure. Similarly, no quantification over variables of an inductive sort is allowed.

In the following, $\mathfrak{I}$ denotes a schematizable class of interpretations and $\mathfrak{F}$ denotes an admissible class of formulæ. The goal of the paper is to prove that if $\mathfrak{I}$-satisfiability is decidable (resp. semi-decidable) for base formulæ in $\mathfrak{F}$ then it must be so for all formulæ in $\mathfrak{F}$. We give examples of classes of formulæ satisfying the previous conditions:

Example 3. Assume that Σ only contains 0, succ and symbols of profile `nat` $\to$ `bool`. Let $\mathfrak{I}_0$ be the class of all $\mathfrak{R}$-compatible interpretations on this language with the usual interpretation of `nat`, 0 and succ, and let $\mathfrak{F}_0$ be the set of all quantifier-free formulæ containing no occurrence of 0 and succ. Clearly, $\mathfrak{I}_0$ is schematizable and $\mathfrak{F}_0$ is admissible. The formulæ in $\mathfrak{F}_0$ denote schemata of propositional formulæ. For instance the schema $p_0 \wedge \neg p_N \wedge \bigwedge_{K=0}^{N-1}(\neg p_K \vee p_{\mathrm{succ}(K)})$ is specified by the formulæ: $p(0) \wedge \neg p(N) \wedge d_N$, where d is defined by the rules $d_0 \to \top$ and $d_{\mathrm{succ}(K)} \to d_K \wedge (\neg p(K) \vee p(\mathrm{succ}(K)))$. $\mathfrak{F}_0$ is equivalent to the class of *regular schemata* in [3].

Example 4. Let $\mathcal{S} = \{\texttt{nat}, \texttt{int}\}$ and $\mathcal{I} = \{\texttt{nat}\}$. Assume that Σ contains the symbols 0 and succ, constant symbols of sort $\texttt{int}$, function symbols of profile $\texttt{nat} \to \texttt{int}$ and all the symbols of Presburger arithmetic. Let $\mathfrak{I}_{\mathbb{Z}}$ be the class of all $\mathfrak{R}$-compatible interpretations such that the interpretations of $\texttt{nat}, \texttt{int}, 0, \mathrm{succ}, +, \leq, \ldots$ are the usual ones. Let $\mathfrak{F}_{\mathbb{Z}}$ be the set of all formulæ built on this language, containing no occurrence of 0, succ, and satisfying Condition 3 in Definition 7. It can be easily checked that $\mathfrak{I}_{\mathbb{Z}}$ is schematizable and that $\mathfrak{F}_{\mathbb{Z}}$ is admissible. Formulæ in $\mathfrak{F}_{\mathbb{Z}}$ denote schemata of Presburger formulæ (the base formulæ in $\mathfrak{F}_{\mathbb{Z}}$ are formulæ of Presburger arithmetic). For instance $\bigvee_{K=0}^{N} a(K) > 0$ is denoted by d_M, with the rules $d_0 \to (a(0) > 0)$ and $d_{\mathrm{succ}(K)} \to d_K \vee a(\mathrm{succ}(K)) > 0$. Note however, that schemata containing atoms with several distinct terms of sort $\texttt{nat}$, such as $\bigwedge_{K=0}^{N} a(K) \simeq a(\mathrm{succ}(K))$ cannot occur in $\mathfrak{F}_{\mathbb{Z}}$. It is also important to remark that the sort $\texttt{int}$ *must* be distinct from the sort of the indices $\texttt{nat}$ (terms of the form $d_{a(K)}$ are *not* allowed).

The class $\mathfrak{F}_{\mathbb{Z}}$ is not comparable to the class of SMT-schemata in [4] (the latter class may contain formulæ of the previous form, at the cost of additional restrictions on the considered theory). Let $\mathfrak{I}_1$ and $\mathfrak{F}_1$ be the sets of interpretations and formulæ fulfilling the conditions of Definitions 6 and 7. The following proposition is easy to establish ($\mathfrak{F}_0$ and $\mathfrak{F}_{\mathbb{Z}}$ are defined in Examples 3 and 4):

Proposition 1. *$\mathfrak{I}_0$-satisfiability (resp. $\mathfrak{I}_{\mathbb{Z}}$-satisfiability) is decidable for base formulæ in $\mathfrak{F}_0$ (resp. $\mathfrak{F}_{\mathbb{Z}}$), and $\mathfrak{I}_1$-satisfiability is semi-decidable for base formulæ in $\mathfrak{F}_1$.*

Before describing the proof procedure for testing the satisfiability of schemata, we provide a simple example of an application. It is only intended to give a taste of what can be expressed in our logic, and of which properties are outside its scope (see also the examples in the Introduction, that can be easily encoded).

Example 5. A (binary) DAG δ labeled by elements of type $\texttt{elem}$ can be denoted by a function symbol $\delta : \texttt{DAG} \to \texttt{elem}$, where the signature contains two constructors of sort $\texttt{DAG}$: a constant symbol $\perp$ (denoting the empty DAG), and a 3-ary symbol $\mathrm{c}(n, l, r)$, where l and r denote the left and right children respectively and n denotes the current node[2]. Various properties can be expressed in our logic, for instance the following defined symbol $A_x^{\delta,p}$ expresses the fact that all the elements occurring in a DAG δ satisfies some property p.

$$A_{\perp}^{\delta,p} \to \top \qquad A_{\mathrm{c}(n,l,r)}^{\delta,p} \to A_l^{\delta,p} \wedge A_r^{\delta,p} \wedge p(\delta(\mathrm{c}(n,l,r)))$$

Obviously this can be generalized to any set of regular positions: for instance, we can state that there exists a path from the root to a leaf in the DAG on which all the element satisfy p:

$$E_{\perp}^{\delta,p} \to \top \qquad E_{\mathrm{c}(n,l,r)}^{\delta,p} \to (E_l^{\delta,p} \vee E_r^{\delta,p}) \wedge p(\delta(\mathrm{c}(n,l,r)))$$

δ and p are meta-variables: δ must be replaced by a function symbol of profile $\texttt{DAG} \to \texttt{elem}$ and p can be replaced by any property of elements of sort $\texttt{elem}$ (provided it

[2] This extra-argument is necessary to ensure that distinct nodes can have the same children.

is expressible in the base language e.g. first-order logic). For instance, we can express the fact that all the elements of δ are equal to some fixed value, or that all the elements of δ are even. We can check that the following formula is valid: $(\forall x, p(x) \Rightarrow q(x)) \Rightarrow (E^{\delta,p} \Rightarrow E^{\delta,q})$. However, the converse *cannot* be expressed in our setting, because it would involve a quantification over an element of type `DAG` which is forbidden by Condition 2 in Definition 7. The formula $A^{\delta,p} \wedge \neg A^{\delta,q} \wedge \neg A^{\delta,\neg q}$ is satisfiable on the interpretations whose domain contains two elements e_1, e_2 such that $p(e_1), p(e_2)$, $\neg q(e_1)$, and $q(e_2)$ hold (but for instance it is unsatisfiable if $p(x) \equiv (x \simeq 0)$). We can express the fact that two DAGs δ and δ' share an element: $\exists x, \forall y, (p(y) \Leftrightarrow x = y) \wedge \neg A^{\delta,\neg p} \wedge \neg A^{\delta',\neg p}$. We can also define a symbol $\mathrm{Map}^{\delta,\delta',f}$ stating that δ' is obtained from δ by applying some function f on every element of δ:

$$\mathrm{Map}_{\perp}^{\delta,\delta',f} \rightarrow \top$$
$$\mathrm{Map}_{\mathrm{c}(n,l,r)}^{\delta,\delta',f} \rightarrow \mathrm{Map}_{l}^{\delta,\delta',f} \wedge \mathrm{Map}_{r}^{\delta,\delta',f} \wedge \delta'(\mathrm{c}(n,l,r)) = f(\delta(\mathrm{c}(n,l,r)))$$

Then, we can check, for instance, that if all the elements of δ are even and if f is the successor function, then all the elements of δ' must be odd:

$$(\mathit{even}(0) \wedge (\forall x,\ \mathit{even}(\mathrm{succ}(x)) \Leftrightarrow \neg \mathit{even}(x)) \wedge A_A^{\delta,\mathit{even}}) \wedge \mathrm{Map}^{\delta,\delta',\mathrm{succ}} \Rightarrow A_A^{\delta',\neg \mathit{even}}$$

We are not able, however, to express transformations affecting the *shape* of the DAG (e.g. switching all the right and left subgraphs) because this would require to use non-monadic defined symbols.

$\mathrm{Alt}^{\delta,p,q}$ expresses the fact that all the elements at even positions satisfy p and that the elements at odd positions satisfy q:

$$\mathrm{Alt}_{\perp}^{\delta,p,q} \rightarrow \top \quad \mathrm{Alt}_{\mathrm{c}(n,l,r)}^{\delta,p,q} \rightarrow \mathrm{Alt}_{l}^{\delta,q,p} \wedge \mathrm{Alt}_{r}^{\delta,q,p} \wedge p(\delta(\mathrm{c}(n,l,r)))$$

Our procedure can be used to verify that $\mathrm{Alt}_A^{\delta,p,q} \Rightarrow A_A^{\delta,p \vee q}$. The following defined symbol $p^{\delta,\delta',\delta''}$ states that a DAG δ'' is constructed by taking elements from δ and δ' alternatively:

$$p_{\perp}^{\delta,\delta',\delta''} \rightarrow \top$$
$$p_{\mathrm{c}(n,l,r)}^{\delta,\delta',\delta''} \rightarrow p_{l}^{\delta',\delta,\delta''} \wedge p_{r}^{\delta',\delta,\delta''} \wedge \delta''(\mathrm{c}(n,l,r)) = \delta(\mathrm{c}(n,l,r))$$

We can check that if the elements of δ and δ' satisfy Properties p and q respectively, then the elements in δ'' satisfy p and q alternatively: $(p_A^{\delta,\delta',\delta''} \wedge A_A^{\delta,p} \wedge A_A^{\delta',q}) \Rightarrow \mathrm{Alt}_A^{\delta'',p,q}$.

Notice that, in this example, the subgraphs can share elements. Thus it is not possible in general to reason independently on each branch (in the style of automata-based approaches): one has to reason *simultaneously* on the whole DAG. Other data structures such as arrays or lists can be handled in a similar way. An example of property that *cannot* be expressed is sortedness. Indeed, it would be stated as follows:

$$\mathrm{Sort}_{\mathrm{c}(n,l,r)}^{\delta} \rightarrow \mathrm{Sort}_{l}^{\delta} \wedge \mathrm{Sort}_{r}^{A} \wedge \delta(\mathrm{c}(n,l,r)) \geq \delta_l \wedge \delta(\mathrm{c}(n,l,r)) \geq \delta_r$$

However, the atom $\delta(\mathrm{c}(n,l,r)) \geq \delta_l$ is *not* allowed in our setting: since it contains several parameters, it contradicts Condition 3 in Definition 7.

3 Proof Procedure

In this section, we present our procedure for testing the $\mathfrak{I}$-satisfiability of admissible formulæ. We employ a tableaux-based procedure, with several kinds of

inference rules: *Decomposition rules* that reduce each formula to a conjunction of base formulæ, equational literals, and defined literals; *Unfolding rules* that allow to unfold the defined atoms (by applying the rules in $\mathfrak{R}$); *Equality rules* for reasoning on equational atoms; and *Delayed instantiation schemes* that replace a parameter A by some term $f(B_1, \ldots, B_n)$, where f is a constructor and $B_1, \ldots, B_n$ are new constant symbols. We consider proof trees labeled by sets of formulæ. If α is a node in a tree $\mathcal{T}$ then $\mathcal{T}(\alpha)$ denotes the label of α. A node is *closed* if it contains $\bot$. As usual, our procedure is specified by a set of *expansion rules* of the form $\dfrac{\Psi}{\Psi_1 \mid \ldots \mid \Psi_n}$ with $n \geq 1$, meaning that a non-closed leaf node labeled by a set $\Phi \supseteq \Psi$ (up to a substitution of the meta-variables) may be expanded by adding n children labeled by $(\Phi \setminus \Psi) \cup \Psi_1, \ldots, (\Phi \setminus \Psi) \cup \Psi_n$ respectively. We assume moreover that the formulæ $\Psi_1, \ldots, \Psi_n$ have not already been generated in the considered branch (to avoid redundant applications of the rules). For any tree $\mathcal{T}$, we write $\alpha \geq_{\mathcal{T}} \beta$ iff β is a child of α. $\geq_{\mathcal{T}}^*$ denotes as usual the reflexive and transitive closure of $\geq_{\mathcal{T}}$.

We need to introduce some additional notations and definitions. For any interpretation I and for any element v in the domain of I, we denote by $\mathrm{depth}_I(v)$ the depth of the constructor term denoted by v, formally defined as follows: $\mathrm{depth}_I(v) = 0$ if v is in $D_{\mathtt{s}}$ and $\mathtt{s} \notin \mathcal{I}$, otherwise $\mathrm{depth}_I([f(t_1, \ldots, t_n)]^I) = 1 + \max(\{\mathrm{depth}_I([t_i]^I) \mid i \in [1, n]\})$, with the convention that $\max(\emptyset) = 0$. It is easy to check that the function $v \mapsto \mathrm{depth}_I(v)$ is well-defined, for every interpretation $I \in \mathfrak{I}$.

For the sake of readability, we shall assume that there exists a function symbol *depth* such that: $\mathit{depth}^I(v) \stackrel{\text{def}}{=} \mathrm{depth}_I(v)$. The formula $\mathit{max}(E) \simeq t$ (where E is a finite set of terms) is written as a shorthand for $\bigwedge_{s \in E}(s \leq t) \wedge \bigvee_{s \in E}(s \simeq t)$ if $E \neq \emptyset$ and for $0 \simeq t$ if $E = \emptyset$.

Let $\mathcal{T}$ be a tree and let α be a node in $\mathcal{T}$. A parameter A is *solved* in α if the only formula of $\mathcal{T}(\alpha)$ containing A is of the form $A \simeq B$ where B is a parameter. An equation $A \simeq B$ is *solved* in α if A is solved. Notice that $\simeq$ is *not* considered as commutative. For every set of formulæ Φ, $\mathrm{Eq}(\Phi)$ denotes the set of equations in Φ and $\mathrm{NonEq}(\Phi) \stackrel{\text{def}}{=} \Phi \setminus \mathrm{Eq}(\Phi)$. A *renaming* is a function ρ mapping every parameter to a parameter of the same sort, such that $\rho(N) = N$. Any renaming ρ can be extended into a function mapping every formula ϕ to a formula $\rho(\phi)$, obtained by replacing every parameter A occurring in ϕ by $\rho(A)$. Let Φ and Ψ be two sets of formulæ. We write $\Phi \sqsupseteq \Psi$ iff there exists a renaming ρ such that $\rho(\Psi) \subseteq \Phi$.

A *proof tree* for ϕ is a tree constructed by the rules of Figure 1 below and such that the root is obtained by applying Start on ϕ. We assume that $\vee$-Decomposition and $\wedge$-Decomposition are applied with the highest priority.

Most of the rules in in Figure 1 are self-explanatory. We only briefly comment on some important points.

Start is only applied once, in order to create the root node of the tree. The label of this node contains the formula at hand together with an additional

formula stating that the max of the depth of the constructor terms represented by the parameters must equal to some natural number N.

The decomposition and closure rules are standard. However, we do *not* use them to test the satisfiability of the formula, but only to decompose it into a conjunction of defined atoms, equational literals and base formulæ. This is always feasible, thanks to the particular properties of formulæ in $\mathfrak{F}$ (see Definition 7). Notice that the separation rule has no premises. The only requirement is that A and B occur in the considered branch.

UNFOLDING replaces a defined atom d_A by its definition according to the rules in $\mathfrak{R}$. This is possible only when the head symbol and arguments of the term represented by A are known.

$\simeq$-DECOMPOSITION decomposes equalities, using the specific properties of $\simeq$-decomposable interpretations: if a node contains two equations $A \simeq t$ and $A \simeq s$ then the formula $\Delta(t \simeq s)$ necessarily holds. $\not\simeq$-DECOMPOSITION performs a similar task for inequalities.

Several rules are introduced to reason on the depth of the terms represented by the parameters. The principle is to separate the parameters representing terms of a depth exactly equal to N from those whose depth is strictly less than N (so that only the former ones may be instantiated). By definition of START, the initial node must contain an equation $\mathit{depth}(A) \preceq N$ for each parameter $A \neq N$. STRICTNESS expands this inequality by using the equivalence $x \preceq y \Leftrightarrow (x \prec y \vee x \simeq y)$. Then $\vee$-DECOMPOSITION will apply, yielding either $x \prec y$ or $x \simeq y$. $\prec$-DECOMPOSITION gets rid of strict equalities of the form $\mathit{depth}(A) \prec \mathrm{succ}(t)$ that are introduced by N-EXPLOSION.

The Explosion rules instantiate the parameters, which is done by adding equations of the form $A \simeq f(\boldsymbol{B})$, where $\boldsymbol{B}$ is a vector of fresh parameters.

EXPLOSION instantiates the parameters distinct from N. We choose to instantiate only the parameters representing terms of maximal depth, and only after N has been instantiated. Thus we instantiate a parameter B only if there exists an atom of the form $\mathit{depth}(B) \simeq t$, where t is of the form $\mathrm{succ}(s)$, for some $s \in \{0, N\}$. EXPLOSION enables further applications of UNFOLDING, which in turn may introduce new complex formulæ into the nodes (by unfolding the defined symbols according to the rules in $\mathfrak{R}$).

N-EXPLOSION instantiates the parameter N. Since the depth of the terms of a sort in $\mathcal{I}$ is at least 1 and since N is intended to denote the maximal depth of the parameters, N cannot be 0, thus it is instantiated either by $\mathrm{succ}(0)$ or by $\mathrm{succ}(N)$. Unlike the other parameters, direct replacement is performed. This rule is applied with the lowest priority. Hence, when the rule is applied, all parameters of a depth strictly greater than N must have been instantiated. By replacing N by a term of the form $\mathrm{succ}(t)$, the rule will permit to instantiate the parameters of depth $N - 1$. This strategy ensures that the parameters will be instantiated in decreasing order w.r.t. the depth of the terms they represent.

LOOP is intended to detect cycles and prune the corresponding branches, by closing the nodes that are subsumed by a previous one. It only applies on some particular nodes, that are irreducible w.r.t. all rules, except (possibly) N-

Explosion. We shall call any such node a *layer*. This rule can be viewed as an application of the induction principle. If $\Phi \sqsupseteq \Psi$ then it is clear that Ψ is a logical consequence of Φ, up to a renaming of parameters. Thus, if some open node exists below a node labeled by Φ, some other open node must exist also below a node labeled by Ψ, hence the node corresponding to Φ may be closed without threatening soundness (a satisfiable branch is closed, but global satisfiability is preserved). Since Ψ is a layer, the parameter N must be instantiated at least once between the two nodes, which ensures that the reasoning is well-founded and that there exists at least one open node outside the branch of Φ.

At first glance, it may seem odd to remove equations from Φ and Ψ before testing for subsumption (see the application condition of Loop). Indeed, it is clear that this operation does *not* preserve satisfiability in general. For instance, the formula $p(A) \wedge \neg p(B) \wedge d_B \wedge A \simeq 0$ is unsatisfiable if d is defined by the rules: $d_0 \rightarrow \top$ and $d_{\mathrm{succ}(K)} \rightarrow \bot$. However, $p(A) \wedge \neg p(B) \wedge d_B$ is satisfiable (with $A^I \neq 0$). In the context in which the rule is applied however, it will be ensured that satisfiability is preserved. The intuition is that if an equation such as $A \simeq 0$ occurs in the node, then A must have been instantiated previously, hence the term represented by A must be of a depth strictly greater than N. Due to the chosen instantiation strategy, all parameters of depth greater or equal to that of A, must have been instantiated (this property is not fulfilled by the previous formula: B should be instantiated since its depth is at most 1 by definition). Then it may be seen that the interpretation of the remaining formulæ does not depend on the value of A, since the depth of their indices must be strictly less than that of A. Note that the removal of equations is *essential* for ensuring termination.

We provide a simple example to illustrate the rule applications.

Example 6. Consider the formula $\forall x\, \neg p(x) \wedge d_A$, together with the rules: $d_a \rightarrow p(b)$ and $d_{f(x,y)} \rightarrow d_x \wedge d_y$ (where $\mathcal{C} = \{a{:}\mathtt{s}, \mathtt{f}{:}\mathtt{s}\times\mathtt{s} \rightarrow \mathtt{s}, \mathtt{0}, \mathrm{succ}\}$ and $\mathit{profile}(A) = \mathtt{s}$). The root formula is $\forall x\, \neg p(x) \wedge d_A \wedge \max(\{\mathit{depth}(A)\}) \simeq N$. By normalization using $\wedge$-Decomposition we get $\{\forall x\, \neg p(x), d_A, \mathit{depth}(A) \simeq N\}$. No rule applies, except N-Explosion, which replaces N by $\mathrm{succ}(0)$ or $\mathrm{succ}(N)$. In both cases, Explosion applies on A. In the first branch, the rule adds the formula $A \simeq a$ and in the second one, it yields $A \simeq f(B, C)$ (where B, C are fresh parameters). In the former branch, Unfolding replaces the formula d_A by $p(b)$, then an irreducible node is reached. In the latter branch, the formulæ d_B and d_C are inferred. Then Loop applies, using the renaming: $\rho(A) = B$ or $\rho(A) = C$, hence the node is closed. The only remaining (irreducible) node is $\{p(b), \forall x\, \neg p(x)\}$. The unsatisfiability of this set of formulæ can be easily checked.

The following example shows evidence of the importance of the depth rules:

Example 7. Consider the formula: $p(A) \wedge d_A \wedge c_B$ with the rules $d_{\mathrm{succ}(x)} \rightarrow d_x, d_0 \rightarrow \top$, $c_{\mathrm{succ}(x)} \rightarrow \bot$ and $c_0 \rightarrow \neg p(0)$. If the parameters were instantiated in an arbitrary order, then one could choose for instance to instantiate A by $\mathrm{succ}(A')$, yielding an obvious loop (indeed, the unfolding of d_A yields $d_{A'}$, thus it suffices to consider the renaming $\rho(A) = A'$ and $\rho(B) = B$). Then the only remaining branch corresponds to the case $A \simeq 0$, which is actually unsatisfiable. This trivial but instructive example shows that reasoning on the depth of the parameters is necessary to ensure that the model will eventually be reached. In this example, the depth of A is maximal and that of B is

not, e.g.: $A \simeq \mathrm{succ}(0)$ and $B \simeq 0$. The problem stems from the fact that LOOP is *not* sound in general, since equational atoms are removed from the formulæ before testing for subsumption (the removal of such atoms is *crucial* for termination).

4 Properties of the Proof Procedure

This short section merely contains the theorems formalizing the main properties of the proof procedure. Due to space restrictions, the proofs are omitted and can be found in [13]. We first state that the previous rules are sound.

Theorem 1. *Let $\mathcal{T}$ be a proof tree for a formula ϕ. If $\mathcal{T}$ is closed then ϕ is unsatisfiable.*

We then state that the procedure is complete, in the sense that the satisfiability of every irreducible node can be tested by the procedure for base formulæ.

Theorem 2. *Let $\mathcal{T}$ be a proof tree. If α is a node in $\mathcal{T}$ that is irreducible by all the expansion rules then $\mathcal{T}(\alpha)$ is $\mathfrak{I}$-satisfiable iff $\mathrm{NonEq}(\mathcal{T}(\alpha))$ is. Furthermore, $\mathrm{NonEq}(\mathcal{T}(\alpha))$ is a set of base formulæ.*

We finally state that the procedure is terminating.

Theorem 3. *The expansion rules terminate on every formula in $\mathfrak{F}$.*

Corollary 1. *If the satisfiability problem is decidable (resp. semi-decidable) for base formulæ in $\mathfrak{F}$ then it is so for all formulæ in $\mathfrak{F}$.*

5 Conclusion

We have proposed a proof procedure for reasoning on schemata of formulæ (defined by induction on an arbitrary structure, such as natural numbers, lists, trees etc.) by relating the satisfiability problem for such schemata to that of a *finite* disjunction of formulæ in the base language. Our approach applies to a wide range of formulæ, which may be interpreted in some specific class of structures (e.g. arithmetics). It may be seen as a generic way to add inductive capabilities into logical languages, in such a way that the main computational properties of the initial language (namely decidability or semi-decidability) are preserved. To the best of our knowledge, no published procedure offers similar features. There are very few decidability or even completeness results in inductive theorem proving and we hope that the present work will help to promote new progress in this direction. Future work includes the implementation of the proof procedure and its extension to non-monadic defined symbols.

$$\textsc{Start:}\ \frac{}{\phi, \max(\{depth(A_i) \mid i \in [1,n]\}) \simeq N}$$ *Where ϕ denotes the formula at hand $A_1, \ldots, A_n$ are the parameters in ϕ*

$$\vee\text{-}\textsc{Decomposition:}\ \frac{\phi \vee \psi}{\phi \mid \psi} \qquad \wedge\text{-}\textsc{Decomposition:}\ \frac{\phi \wedge \psi}{\phi, \psi}$$ *If $\phi \wedge \psi$ is not a base formula*

$$\textsc{Closure:}\ \frac{\neg\phi, \phi}{\bot} \qquad \simeq\text{-}\textsc{Closure:}\ \frac{A \not\simeq A}{\bot} \qquad N\text{-}\textsc{Closure:}\ \frac{0 \simeq \mathrm{succ}(t)}{\bot}$$

$$\textsc{Unfolding:}\ \frac{d_A, A \simeq f(\boldsymbol{B})}{\psi} \qquad \frac{\neg d_A, A \simeq f(\boldsymbol{B})}{NNF(\neg\psi)} \qquad \psi = d_{f(\boldsymbol{B})}{\downarrow_{\mathfrak{R}}}\,[A/f(\boldsymbol{B})], A \simeq f(\boldsymbol{B})$$

$$\simeq\text{-}\textsc{Decomposition:}\ \frac{A \simeq f(\boldsymbol{B}), A \simeq g(\boldsymbol{C})}{\psi, A \simeq f(\boldsymbol{B})} \qquad \frac{A \not\simeq B, A \simeq f(\boldsymbol{B}), B \simeq g(\boldsymbol{C})}{NNF(\neg\psi), A \not\simeq B, A \simeq f(\boldsymbol{B}), B \simeq g(\boldsymbol{C})}$$

Where $\psi = \Delta(f(\boldsymbol{B}) \simeq g(\boldsymbol{C}))$[a]

$$\textsc{Replacement:}\ \frac{\phi, A \simeq B}{\phi[B/A], A \simeq B}$$ *If A and B are two parameters and A occurs in ϕ*

$$\textsc{Strictness:}\ \frac{depth(A) \preceq N}{depth(A) \simeq N \vee depth(A) \prec N} \qquad \prec\text{-}\textsc{Decomposition:}\ \frac{t \prec \mathrm{succ}(N)}{t \preceq N}$$

$$\prec\text{-}\textsc{Separation:}\ \frac{depth(A) \prec N, depth(B) \simeq N}{depth(A) \prec N, depth(B) \simeq N, A \not\simeq B} \qquad \textsc{Separation:}\ \frac{}{A \simeq B \vee A \not\simeq B}$$

$$\textsc{Explosion:}\ \frac{depth(B) \simeq \mathrm{succ}(t)}{\bigvee_{i \in [1,n]} max(E_i) \simeq t \wedge B \simeq t_i}$$

If t_i are terms of the form $f_i(\boldsymbol{A_i})$, such that $f_1, \ldots, f_n$ are all the function symbols of the same sort as B, and the $\boldsymbol{A_i}$'s are vectors of pairwise distinct, fresh, constant symbols of the appropriate sort, and E_i is the set of terms $depth(C)$, where C is a component of $\boldsymbol{A_i}$ of a sort in $\mathcal{I}$.

$$N\text{-}\textsc{Explosion:}\ \frac{\Phi}{\Phi[\mathrm{succ}(0)/N] \mid \Phi[\mathrm{succ}(N)/N]}$$

If no other rule applies and N occurs in Φ. Notice that in contrast with the previous rules, Φ must denote the whole label (not a subset of it)

$$\textsc{Loop:}\ \frac{\Phi}{\bot}$$ *If there exists in the same branch a (non leaf) layer labeled by a set of formulæ Ψ such that* $\mathrm{NonEq}(\Phi) \sqsupseteq \mathrm{NonEq}(\Psi)$

[a] See Definition 4 for the definition of $\Delta(t \simeq s)$

Fig. 1. Expansion rules

References

1. P. Aczel. An Introduction to Inductive Definitions. In K. J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland, Amsterdam, 1977.
2. V. Aravantinos, R. Caferra, and N. Peltier. A schemata calculus for propositional logic. In *TABLEAUX 09 (International Conference on Automated Reasoning with Analytic Tableaux and Related Methods)*, volume 5607 of *LNCS*, pages 32–46. Springer, 2009.
3. V. Aravantinos, R. Caferra, and N. Peltier. Decidability and undecidability results for propositional schemata. *Journal of Artificial Intelligence Research*, 40:599–656, 2011.
4. V. Aravantinos and N. Peltier. Schemata of SMT problems. In *TABLEAUX 11 (International Conference on Automated Reasoning with Analytic Tableaux and Related Methods)*, LNCS. Springer, 2011.
5. M. Baaz, S. Hetzl, A. Leitsch, C. Richter, and H. Spohr. CERES: An analysis of Fürstenberg's proof of the infinity of primes. *Theor. Comput. Sci.*, 403(2-3):160–175, 2008.
6. D. Baelde, D. Miller, and Z. Snow. Focused inductive theorem proving. In *IJCAR*, pages 278–292, 2010.
7. C. Benzmüller, L. C. Paulson, F. Theiss, and A. Fietzke. LEO-II - A Cooperative Automatic Theorem Prover for Classical Higher-Order Logic (System Description). In *Proceedings of the IJCAR'08*, pages 162–170. Springer-Verlag, 2008.
8. A. Bouhoula, E. Kounalis, and M. Rusinowitch. SPIKE, an automatic theorem prover. In *Proceedings of LPAR'92*, volume 624, pages 460–462. Springer-Verlag, 1992.
9. A. Bouhoula and M. Rusinowitch. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 14:14–189, 1995.
10. R. S. Boyer and J. S. Moore. A theorem prover for a computational logic. In M. E. Stickel, editor, *CADE*, volume 449 of *LNCS*, pages 1–15. Springer, 1990.
11. A. Bundy. The automation of proof by mathematical induction. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 845–911. Elsevier and MIT Press, 2001.
12. H. Comon. Inductionless induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 14, pages 913–962. North-Holland, 2001.
13. M. Echenim and N. Peltier. Reasoning on Schemata of Formulae. Technical report, CoRR, abs/1204.2990, 2012.
14. J. Giesl and D. Kapur. Decidable classes of inductive theorems. In R. Goré, A. Leitsch, and T. Nipkow, editors, *IJCAR*, volume 2083 of *LNCS*, pages 469–484. Springer, 2001.
15. A. Gupta and A. L. Fisher. Parametric circuit representation using inductive boolean functions. In C. Courcoubetis, editor, *CAV*, volume 697 of *LNCS*, pages 15–28. Springer, 1993.
16. D. Kapur and D. Musser. Proof by consistency. *Artificial Intelligence*, 31, 1987.
17. G. Lenzi. A New Logical Characterization of Büchi Automata. In A. Ferreira and H. Reichel, editors, *STACS 2001*, volume 2010 of *LNCS*, pages 467–477. Springer Berlin / Heidelberg, 2001.
18. D. M. Park. Finiteness is Mu-ineffable. *Theoretical Computer Science*, 3:173–181, 1976.
19. C. Paulin-Mohring. Inductive Definitions in the system Coq - Rules and Properties. In *TLCA '93*, pages 328–345, London, UK, 1993. Springer-Verlag.