Combining Lists with Non-Stably Infinite Theories

Pascal Fontaine, Silvio Ranise, Calogero Zarba

Combining Lists with Non-Stably Infinite Theories

Pascal Fontaine, Silvio Ranise, and Calogero G. Zarba

LORIA and INRIA-Lorraine

Abstract. In program verification one has often to reason about lists over elements of a given nature. Thus, it becomes important to be able to combine the theory of lists with a generic theory T modeling the elements. This combination can be achieved using the Nelson-Oppen method *only if* T is stably infinite.
The goal of this paper is to relax the stable-infiniteness requirement. More specifically, we provide a new method that is able to combine the theory of lists with any theory T of the elements, regardless of whether T is stably infinite or not. The crux of our combination method is to guess an arrangement over a set of variables that is larger than the one considered by Nelson and Oppen.
Furthermore, our results entail that it is also possible to combine T with the more general theory of lists with a length function.

1 Introduction

In program verification one has often to decide the validity or satisfiability of logical formulae involving lists over elements of a given nature. For instance, these formulae may involve lists of integers or lists of booleans.

One way to reason about lists over elements of a given nature is to use the Nelson-Oppen method in order to modularly combine a decision procedure for a theory modeling lists with a decision procedure for a theory modeling the elements. This solution requires that the theory of the elements be *stably infinite*. Unfortunately, this requirement is not satisfied by many interesting theories such as, for instance, the theory of booleans and the theory of integers modulo n.

In this paper, we show how to relax the stable infiniteness requirement. More specifically, let T_{list} be the two-sorted theory of lists involving a sort elem for elements, a sort list for flat lists of elements, plus the symbols nil, car, cdr, and cons. For instance, a valid formula in T_{list} is

$$x \approx \mathsf{cdr}(\mathsf{cons}(a, \mathsf{nil})) \;\rightarrow\; x \not\approx \mathsf{cons}(b, y)\,.$$

We consider the theory T_{lint} that extends T_{list} with a sort int for the integers, the symbols 0, 1, $+$, $-$, $<$ for reasoning over the integers, and a function symbol length whose arity is $\mathsf{list} \rightarrow \mathsf{int}$. For instance, a valid formula in T_{lint} is

$$x \not\approx \mathsf{cdr}(\mathsf{cons}(a, \mathsf{nil})) \;\rightarrow\; \mathsf{length}(x) > 0\,.$$

We then provide a combination method that is able to combine T_{lint} with any theory T_{elem} modeling the elements, regardless of whether T_{elem} is stably infinite or not.

The core ideas of our combination method are:

- modifying the Nelson-Oppen method in such a way to guess an arrangement over an extended set of free constants, and not just the shared ones.
- opportunely computing a certain minimal cardinality k_0, so that we can ensure that the domain of the elements must have at least k_0 elements.

1.1 Related work

The importance of reasoning about lists is corroborated by the numerous flavors of theories of lists [1, 3, 4, 12, 13, 17] present in literature, as well as by the increasing number of tools [6, 7, 11, 14, 15, 18] containing some capabilities for reasoning about lists.

The idea of guessing an arrangement over a larger sets of free constants was already used by Zarba in order to combine the theory of sets [23] and the theory of multisets [21] with any arbitrary theory T of the elements, regardless of whether T is stably infinite or not. This idea was also used by Fontaine and Gribomont [8] in order to combine the theory of arrays with any other non-necessarily stably infinite theory T.

The idea of computing minimal cardinalities was used by Zarba [22] in order to combine the theory of finite sets with a non-necessarily stably infinite theory T of the elements, in the presence of the cardinality operator. This idea was also exploited by Tinelli and Zarba [19], who provided a method for combining any *shiny* theory S with any non-necessarily stably infinite theory T. Examples of shiny theories include the theory of equality, the theories of partial and total orders, and the theories of lattices with maximum and minimum.

2 Many-sorted logic

2.1 Syntax

We fix the following infinite sets: a set **sorts** of sorts, a set **var** of variables, a set **con** of constant symbols, a set **fun** of functions symbols, and a set **pred** of predicate symbols. We also fix an infinite set **par** of constant symbols disjoint from **con**. We call parameters the elements of **par**.

A *signature* Σ is a tuple $\langle S, C, F, P\rangle$ where $S \subseteq \mathbf{sorts}$, $C \subseteq \mathbf{con} \cup \mathbf{par}$, $F \subseteq \mathbf{fun}$, $P \subseteq \mathbf{pred}$, all the symbols in C have sorts in S, and all the symbols in F, P have arities constructed using the sorts in S. If $\Sigma = \langle S, C, F, P\rangle$ is a signature, we sometimes write Σ^{S} for S, Σ^{C} for C, Σ^{F} for F, and Σ^{P} for P.

If $\Sigma_1 = \langle S_1, C_1, F_1, P_1\rangle$ and $\Sigma_2 = \langle S_2, C_2, F_2, P_2\rangle$ are signatures, we write $\Sigma_1 \subseteq \Sigma_2$ when $S_1 \subseteq S_2$, $C_1 \subseteq C_2$, $F_1 \subseteq F_2$, and $P_1 \subseteq P_2$. If $\Sigma_1 = \langle S_1, C_1, F_1, P_1\rangle$ and $\Sigma_2 = \langle S_2, C_2, F_2, P_2\rangle$ are signatures, their *union* is the signature $\Sigma_1 \cup \Sigma_2 = \langle S_1 \cup S_2, C_1 \cup C_2, F_1 \cup F_2, P_1 \cup P_2\rangle$. Let $\Sigma = \langle S, C, F, P\rangle$ be a signature, and

let C_0 be a set of constant symbols. We denote with $\Sigma(C_0)$ the signature $= \langle S, C \cup C_0, F, P\rangle$.

Given a signature Σ, we assume the standard notions of Σ-term, Σ-atom, Σ-literal, Σ-formula. Σ-sentences are Σ-formulae with no free variables.

If φ is either a term or a formula, we denote with $pars_\sigma(\varphi)$ the set of parameters of sort σ occurring in φ. If φ is either a term or a formula, we denote with $pars(\varphi)$ the set $\bigcup_{\sigma \in \mathbf{sorts}} pars_\sigma(\varphi)$.

In the rest of this paper we identify conjunction of formulae $\varphi_1 \wedge \cdots \wedge \varphi_n$ with the set $\{\varphi_1, \ldots, \varphi_n\}$. In addition, we abbreviate literals of the form $\neg(s \approx t)$ with $s \not\approx t$.

2.2 Semantics

Definition 1. If Σ is a signature, a Σ-STRUCTURE $\mathcal{A}$ is a map which interprets:[1]

- each sort $\sigma \in \Sigma^{\mathrm{S}}$ as a non-empty domain A_σ;
- each variable $x \in X$ of sort σ as an element $x^{\mathcal{A}} \in A_\sigma$;
- each constant symbol $c \in \Sigma^{\mathrm{C}}$ of sort σ as an element $c^{\mathcal{A}} \in A_\sigma$;
- each function symbol $f \in \Sigma^{\mathrm{F}}$ of arity $\sigma_1 \times \cdots \times \sigma_n \to \tau$ as a function $f^{\mathcal{A}} : A_{\sigma_1} \times \cdots \times A_{\sigma_n} \to A_\tau$;
- each predicate symbol $p \in \Sigma^{\mathrm{P}}$ of arity $\sigma_1 \times \cdots \times \sigma_n$ as a subset $P^{\mathcal{A}}$ of $A_{\sigma_1} \times \cdots \times A_{\sigma_n}$. □

A Σ-sentence φ is *satisfiable* if it evaluates to true under some Σ-structure.

Let $\mathcal{A}$ be an Ω-structure, and let $\Sigma \subseteq \Omega$. We denote with $\mathcal{A}^\Sigma$ the structure obtained from $\mathcal{A}$ by restricting it to interpret only the symbols in Σ.

2.3 Theories

Following Ganzinger [9], we define theories as sets of structures rather than as sets of sentences. More formally, we give the following definition.

Definition 2. A Σ-THEORY is a pair $\langle \Sigma, \mathbf{A}\rangle$ where Σ is a signature such that $\Sigma^{\mathrm{C}} \cap \mathbf{par} = \emptyset$, and $\mathbf{A}$ is a set of Σ-structures. □

Definition 3. Let T be a Σ-theory, and let $\Sigma \subseteq \Omega$. We say that an Ω-structure $\mathcal{A}$ is a T-STRUCTURE if $\mathcal{A}^\Sigma \in T$. □

A sentence φ is *T-satisfiable* if it evaluates to true under some T-structure.

Given a Σ-theory, the *ground satisfiability problem* of T is the problem of deciding, for each ground $\Sigma(\mathbf{par})$-formula φ, whether or not φ is T-satisfiable.

Definition 4. Let Σ be a signature, let $S \subseteq \Sigma^{\mathrm{S}}$ be a nonempty set of sorts, and let T be a Σ-theory. We say that T is STABLY INFINITE with respect to S if every ground $\Sigma(\mathbf{par})$-formula φ is T-satisfiable if and only if there exists a T-structure satisfying φ such that A_σ is infinite, for each sort $\sigma \in S$. □

[1] Unless otherwise specified, we use the convention that calligraphic letters denote structures, and that the corresponding Roman letters, opportunely subscripted, denote the domains of the structures.

Definition 5 (Combination of theories). Let $T_i = \langle \Sigma_i, \mathbf{A}_i \rangle$ be a theory, for $i = 1, 2$. The COMBINATION of T_1 and T_2 is the theory $comb(T_1, T_2) = \langle \Sigma, \mathbf{A} \rangle$ where $\Sigma = \Sigma_1 \cup \Sigma_2$ and $\mathbf{A} = \{\mathcal{A} \mid \mathcal{A}^{\Sigma_1} \in \mathbf{A}_1 \text{ and } \mathcal{A}^{\Sigma_2} \in \mathbf{A}_2\}$. □

2.4 The theory of integers

Let us fix a signature Σ_{int} containing a sort int for the integers, plus the constant symbols 0 and 1 of sort int, the function symbols $+$ and $-$ of arity $\mathsf{int} \to \mathsf{int}$, and the predicate symbol $<$, of arity $\mathsf{int} \times \mathsf{int}$.

Definition 6. The STANDARD int-STRUCTURE is the Σ_{int}-structure $\mathcal{A}$ specified by letting $A_{\mathsf{int}} = \mathbb{Z}$ and interpreting the symbols $0, 1, +, -, <$ according to their intuitive meaning over $\mathbb{Z}$. □

Definition 7. The THEORY OF INTEGERS is the pair $T_{\mathsf{int}} = \langle \Sigma_{\mathsf{int}}, \{\mathcal{A}\} \rangle$, where $\mathcal{A}$ is the standard int-structure. □

The ground satisfiability problem of T_{int} can be decided by using methods based on integer automata [20], the omega test [2, 16], or opportune extensions of the Fourier-Motzkin method [10].

2.5 Lists

Let A be a non-empty set, and assume that the special object $\perp$ does not belong to A. A *list* x over A of *length* n is a map $x : \mathbb{N} \to A \cup \{\perp\}$ such that $x(i) \in A$, for $i < n$, and $x(i) = \perp$, for $i \geq n$. We write $|x| = n$ to indicate that the length of the list x is n. We denote with A^* the set of lists over A.

We denote with *nil* the empty list, that is, $nil(i) = \perp$, for each $i \in \mathbb{N}$. We denote with *car* and *cons* the partial functions defined as follows: given a list $x \neq nil$, we let $car(x) = x(0)$, whereas $cdr(x)$ is the unique list y such that $y(n) = x(n + 1)$, for each $n \in \mathbb{N}$.

Given an element $e \in A$ and a list x in A^*, we denote with $cons(e, x)$ the list y such that $y(0) = e$, and $y(n + 1) = x(n)$, for each $n \in \mathbb{N}$.

2.6 The theory of lists

We fix a signature Σ_{list} containing a sort elem for elements and a sort list for lists of elements, plus the constant symbol $\perp_{\mathsf{elem}}$ of sort elem, the constant symbols nil and $\perp_{\mathsf{list}}$ of sort list, the function symbols car of arity $\mathsf{list} \to \mathsf{elem}$, the function symbol cdr of arity $\mathsf{list} \to \mathsf{list}$, and the function symbol cons of arity $\mathsf{elem} \times \mathsf{list} \to \mathsf{list}$.

Definition 8. A STANDARD list-STRUCTURE $\mathcal{A}$ is a Σ_{list}-structure satisfying the following conditions:

- $\perp \notin A_{\mathsf{elem}}$;
- $A_{\mathsf{list}} = (A_{\mathsf{elem}})^*$;

– $\mathsf{nil}^{\mathcal{A}} = nil$;
– $\mathsf{car}^{\mathcal{A}}(nil) = (\bot_{\mathsf{elem}})^{\mathcal{A}}$;
– $\mathsf{cdr}^{\mathcal{A}}(nil) = (\bot_{\mathsf{list}})^{\mathcal{A}}$;
– $\mathsf{car}^{\mathcal{A}}(x) = car(x)$, for each $x \in A_{\mathsf{list}}$ such that $x \neq nil$;
– $\mathsf{cdr}^{\mathcal{A}}(x) = cdr(x)$, for each $x \in A_{\mathsf{list}}$ such that $x \neq nil$;
– $\mathsf{cons}^{\mathcal{A}}(e, x) = cons(e, x)$, for each $e \in A_{\mathsf{elem}}$ and $x \in A_{\mathsf{list}}$. □

Note that although *car* and *cdr* are partial functions, standard list-structures interpret the symbols car and cdr as total functions. In particular, all standard list-structures ensure that the constants $\bot_{\mathsf{elem}}$ and $\bot_{\mathsf{list}}$ have the same interpretations of the terms car(nil) and cdr(nil), respectively.

Definition 9. The THEORY OF LISTS is the pair $T_{\mathsf{list}} = \langle \Sigma_{\mathsf{list}}, \mathbf{A} \rangle$, where $\mathbf{A}$ is the set of all standard list-structures. □

As a by product of the results of this paper, we will see that the ground satisfiability problem of T_{list} can be decided by opportunely adapting Oppen's decision procedure for a one-sorted theory of lists without nil [13].

2.7 The theory of lists with a length function

We fix a signature Σ_{lint} containing all the symbols in Σ_{int} and Σ_{list}, plus the function symbol length of arity list → int.

Definition 10. A STANDARD lint-STRUCTURE $\mathcal{A}$ is a Σ_{lint}-structure satisfying the following conditions:

– $\mathcal{A}^{\Sigma_{\mathsf{int}}}$ is the standard int-structure;
– $\mathcal{A}^{\Sigma_{\mathsf{list}}}$ is a standard list-structure;
– $\mathsf{length}^{\mathcal{A}}(x) = |x|$, for each $x \in A_{\mathsf{list}}$. □

Definition 11. The THEORY OF LISTS WITH A LENGTH FUNCTION is the pair $T_{\mathsf{lint}} = \langle \Sigma_{\mathsf{lint}}, \mathbf{A} \rangle$, where $\mathbf{A}$ is the set of all standard lint-structures. □

The ground satisfiability problem of T_{lint} can be decided by opportunely adapting a decision procedure for a two-sorted theory of recursively defined data structures with integer constraints [24].

3 The combination method

Let Σ_{elem} be a signature such that $\Sigma^{\mathrm{S}} = \{\mathsf{elem}\}$, and let T_{elem} be any Σ_{elem}-theory, not necessarily stably infinite with respect to the sort elem. Assume that the ground satisfiability problem of T_{elem} is decidable. We now describe a combination-based decision procedure for the ground satisfiability problem of $T = comb(T_{\mathsf{elem}}, T_{\mathsf{lint}})$.

In our combination method we use as black boxes a decision procedure for the ground satisfiability problem of T_{elem} and a decision procedure for the ground

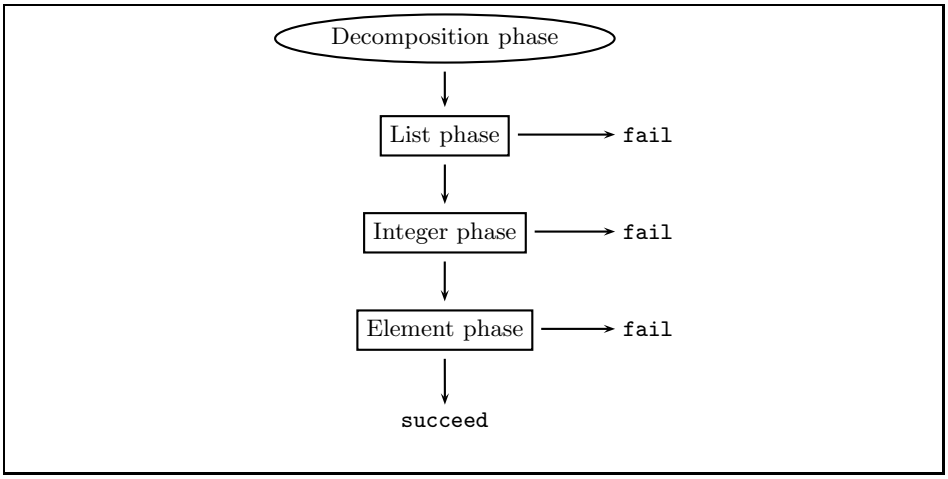

Fig. 1: The phases of our combination method.

satisfiability problem of T_{int}. We also use—albeit not strictly as a black box—Oppen's decision procedure for recursively defined data structures.

Without loss of generality, we restrict ourselves to conjunctions Γ of literals in *separate* form: $\Gamma = \Gamma_{\mathsf{elem}} \cup \Gamma_{\mathsf{int}} \cup \Gamma_{\mathsf{list}} \cup \Gamma_{\mathsf{length}}$ where:

(a) Γ_{elem} contains only $\Sigma_{\mathsf{elem}}(\mathbf{par})$-literals;
(b) Γ_{int} contains only $\Sigma_{\mathsf{int}}(\mathbf{par})$-literals;
(c) Γ_{list} contains only flat $\Sigma_{\mathsf{list}}(\mathbf{par})$-literals of the form

$$x \approx_{\mathsf{list}} y\,, \qquad x \not\approx_{\mathsf{list}} y\,, \qquad x \approx \mathsf{nil}\,,$$
$$e \approx \bot_{\mathsf{elem}}\,, \qquad x \approx \bot_{\mathsf{list}}\,, \qquad x \approx \mathsf{cons}(e, y)\,,$$

where e_1, e_2, e are elem-parameters and x, y are list-parameters.
(d) Γ_{length} contains only literals of the form $u \approx \mathsf{length}(x)$ where u is an int-parameter and x is a list-parameter;
(e) for each list-parameter $x \in \mathit{pars}_{\mathsf{list}}(\Gamma)$, either $x \approx \mathsf{nil}$ or $x \not\approx \mathsf{nil}$ is in Γ_{list}.[2]

Our combination method consists of the four phases depicted in Figure 1, and described below.

3.1 Decomposition phase

Let $\Gamma = \Gamma_{\mathsf{elem}} \cup \Gamma_{\mathsf{int}} \cup \Gamma_{\mathsf{list}} \cup \Gamma_{\mathsf{length}}$ be a conjunction of literals in separate form. Also let $P_{\mathsf{elem}} = \mathit{pars}_{\mathsf{elem}}(\Gamma_{\mathsf{list}}) \cup \{\bot_{\mathsf{elem}}\}$ and $P_{\mathsf{list}} = \mathit{pars}_{\mathsf{list}}(\Gamma)$. In the decomposition

[2] We remind to Section 5 for a possible way of enforcing properties (a)–(e) that involves the introduction of fresh parameters, as well as the employment of a state-of-the-art propositional reasoner for efficiency concerns.

phase we non-deterministically guess an equivalence relation $\sim_{\mathsf{elem}}$ of P_{elem}, and we construct the following set of literals:

$$\alpha_{\mathsf{elem}} = \{e_1 \approx e_2 \mid e_1 \sim_{\mathsf{elem}} e_2\} \cup \{e_1 \not\approx e_2 \mid e_1, \nsim_{\mathsf{elem}} e_2\}\,.$$

Note that our decomposition phase differs from the one of Nelson-Oppen method. In fact, in the Nelson-Oppen one guesses an equivalence relation over the smaller set of parameters $pars_{\mathsf{elem}}(\Gamma_{\mathsf{elem}}) \cap pars_{\mathsf{elem}}(\Gamma_{\mathsf{list}})$. We need to use the larger set P_{elem} because we do not have any stable infiniteness assumption over the theory T_{elem} of the elements.

3.2 List phase

In the list phase we essentially employ Oppen's decision procedure for recursively defined data structures. By not using Oppen's procedure just as a black box, we will later be able to use the information constructed in this phase in the later phases of our method. (Cf. Section 5.)

More in detail, in the list phase we construct the minimal equivalence relation $\sim_{\mathsf{list}}$ of P_{list} satisfying the following conditions:

(a) if $x \approx y$ is in Γ_{list} then $x \sim_{\mathsf{list}} y$;
(b) if $x_1 \approx \mathsf{cons}(e_1, y_1)$ and $x_2 \approx \mathsf{cons}(e_2, y_2)$ are in Γ_{list}, and $e_1 \sim_{\mathsf{elem}} e_2$ and $y_1 \sim_{\mathsf{list}} y_2$ then $x_1 \sim_{\mathsf{list}} x_2$;
(c) if $x_1 \approx \mathsf{cons}(e_1, y_1)$ and $x_2 \approx \mathsf{cons}(e_2, y_2)$ are in Γ_{list}, and $x_1 \sim_{\mathsf{list}} x_2$ then $e_1 \sim_{\mathsf{elem}} e_2$ and $y_1 \sim_{\mathsf{list}} y_2$.

Furthermore, we construct the relation $\prec_{\mathsf{list}}$ of P_{list} defined by letting $x \prec_{\mathsf{list}} y$ if and only if there are list-parameters $x', y' \in P_{\mathsf{list}}$ and an elem-parameter $e \in P_{\mathsf{elem}}$ such that $x \sim_{\mathsf{list}} x'$, $y \sim_{\mathsf{list}} y'$, and the literal $y' \approx \mathsf{cons}(e, x')$ is in Γ_{list}.

We end our method by outputting `fail` if at least one of the following conditions does not hold:

(C1) If $x \sim_{\mathsf{list}} y$ then the literal $x \not\approx_{\mathsf{list}} y$ is not in Γ_{list};
(C2) There are no two literals $x \approx \mathsf{nil}$ and $y \approx \mathsf{cons}(e, z)$ in Γ_{list} for which $x \sim_{\mathsf{list}} y$;
(C3) The relation $\prec_{\mathsf{list}}$ is well-founded.

If instead all conditions (C1)–(C3) hold, we proceed to the next phase.

3.3 Integer phase

In this phase we extract integer constraints from the conjunctions Γ_{list} and Γ_{length}, as well as from the equivalence relation $\sim_{\mathsf{list}}$ constructed in the list phase.

More in detail, we generate a fresh int-parameter u_x, for each list-parameter x in P_{list}, and we construct the following set of literals

$$\begin{aligned}\alpha_{\mathsf{int}} = &\{u_x \approx 0 \mid x \approx \mathsf{nil} \text{ is in } \Gamma_{\mathsf{list}}\} \cup \\ &\{u_x > 0 \mid x \not\approx \mathsf{nil} \text{ is in } \Gamma_{\mathsf{list}}\} \cup \\ &\{u_x = u_y + 1 \mid x \approx \mathsf{cons}(e, y) \text{ is in } \Gamma_{\mathsf{list}}\} \cup \\ &\{u \approx u_x \mid u \approx \mathsf{length}(x) \text{ is in } \Gamma_{\mathsf{length}}\} \cup \\ &\{u_x \approx u_y \mid x \sim_{\mathsf{list}} y\}\,.\end{aligned}$$

Then, we check whether $\Gamma_{\mathsf{int}} \cup \alpha_{\mathsf{int}}$ is T_{int}-satisfiable. If this is not the case, we end our method by outputting `fail`; otherwise we proceed to the next phase.

3.4 Element phase

We will prove later that when we reach this point we can already conclude that $\alpha_{\mathsf{elem}} \cup \Gamma_{\mathsf{list}} \cup \Gamma_{\mathsf{int}} \cup \Gamma_{\mathsf{length}}$ is T_{lint}-satisfiable.[3] Therefore, we can effectively compute the minimal integer k_0 for which there exists a T_{lint}-structure $\mathcal{A}$ satisfying $\alpha_{\mathsf{elem}} \cup \Gamma_{\mathsf{list}} \cup \Gamma_{\mathsf{int}} \cup \Gamma_{\mathsf{length}}$ such that $k_0 = |A_{\mathsf{elem}}|$.[4]

The last step of the element phase consists of checking whether $\Gamma_{\mathsf{elem}} \cup \alpha_{\mathsf{elem}} \cup \{|\mathsf{elem}| \geq k_0\}$ is T_{elem}-satisfiable.[5] If this is not the case, we end the method by outputting `fail`; otherwise we happily output `succeed`.

4 Correctness

In this section we prove that our combination method is correct. Clearly, our method is terminating. The following proposition shows that our method is also partially correct.

Proposition 12. *Let T_{elem} be a Σ_{elem}-theory such that $\Sigma^{\mathrm{S}} = \{\mathsf{elem}\}$, let $T = comb(T_{\mathsf{elem}}, T_{\mathsf{lint}})$, and let $\Gamma = \Gamma_{\mathsf{elem}} \cup \Gamma_{\mathsf{int}} \cup \Gamma_{\mathsf{list}} \cup \Gamma_{\mathsf{length}}$ be a conjunction of literals in separate form. Then the following are equivalent:*

1. *Γ is T-satisfiable.*
2. *There exists an equivalence relation $\sim_{\mathsf{elem}}$ of $pars_{\mathsf{elem}}(\Gamma_{\mathsf{list}}) \cup \{\bot_{\mathsf{elem}}\}$ for which our method outputs* `succeed`. □

Proof. To simplify the notation, we let $P_{\mathsf{elem}} = pars_{\mathsf{elem}}(\Gamma_{\mathsf{list}}) \cup \{\bot_{\mathsf{elem}}\}$ and $P_{\mathsf{list}} = pars_{\mathsf{list}}(\Gamma)$.

[3] A T_{lint}-structure satisfying $\alpha_{\mathsf{elem}} \cup \Gamma_{\mathsf{list}} \cup \Gamma_{\mathsf{int}} \cup \Gamma_{\mathsf{length}}$ is denoted with $\mathcal{C}$ in the second part of the proof of Proposition 12.

[4] This computation could be done in a naive way by enumerating all T_{lint}-structures over the parameters occurring in Γ, in increasing order with respect to the cardinality of the domain of the elements.

[5] With $\{|\mathsf{elem}| \geq k_0\}$ we denote the set of disequalities $\{e_i \not\approx e_j \mid 1 \leq i < j \leq k_0\}$, where the e_i are fresh elem-parameters.

$(1 \Rightarrow 2)$. Let $\mathcal{M}$ be a T-structure satisfying Γ. We define an equivalence relation $\sim_{\mathsf{elem}}$ over P_{elem} by letting

$$e_1 \sim_{\mathsf{elem}} e_2 \iff e_1^{\mathcal{M}} = e_2^{\mathcal{M}}\,, \qquad \text{for each } e_1, e_2 \in P_{\mathsf{elem}}\,.$$

We claim that if we guess $\sim_{\mathsf{elem}}$ as defined above, then our method outputs `succeed`. To see this, let $\sim_{\mathsf{list}}$ be the equivalence relation constructed in the list phase, and let $\equiv_{\mathsf{list}}$ be the equivalence relation of P_{list} defined as follows:

$$x \equiv_{\mathsf{list}} y \iff x^{\mathcal{M}} = y^{\mathcal{M}}\,, \qquad \text{for each } x, y \in P_{\mathsf{list}}\,.$$

By construction $\equiv_{\mathsf{list}}$ satisfies conditions (a)–(c) in the list phase. Therefore, we have $\sim_{\mathsf{list}} \subseteq \equiv_{\mathsf{list}}$, that is:

$$x \sim_{\mathsf{list}} y \Longrightarrow x \equiv_{\mathsf{list}} y\,, \qquad \text{for each } x, y \in P_{\mathsf{list}}\,.$$

By using the fact that $\sim_{\mathsf{list}} \subseteq \equiv_{\mathsf{list}}$, one can verify that $\sim_{\mathsf{list}}$ satisfies all conditions (C1)–(C3) of the list phase. Therefore, our method does not output `fail` when executing the list phase.

Next, we claim that our method also does not output `fail` when executing the integer phase. To justify the claim, we need to show that $\Gamma_{\mathsf{int}} \cup \alpha_{\mathsf{int}}$ is T_{int}-satisfiable. Indeed, by again using the fact that $\sim_{\mathsf{list}} \subseteq \equiv_{\mathsf{list}}$, it is possible to verify that a T_{int}-structure satisfying $\Gamma_{\mathsf{int}} \cup \alpha_{\mathsf{int}}$ can be obtained by extending $\mathcal{M}$ to the parameters u_x by letting

$$u_x^{\mathcal{M}} = |x^{\mathcal{M}}|\,, \qquad \text{for each } \mathsf{list}\text{-parameter } x \in P_{\mathsf{list}}\,.$$

It remains to show that our method outputs `succeed` when executing the element phase. To see this, let k_0 be the minimal integer computed in the element phase. By construction, $\mathcal{M}$ satisfies $\Gamma_{\mathsf{elem}} \cup \alpha_{\mathsf{elem}}$. More over, since $\mathcal{M}$ satisfies $\alpha_{\mathsf{elem}} \cup \Gamma_{\mathsf{list}} \cup \Gamma_{\mathsf{int}} \cup \Gamma_{\mathsf{length}}$, it must have at least k_0 elements. It follows that $\mathcal{M}$ is a T_{elem}-structure satisfying $\Gamma_{\mathsf{elem}} \cup \alpha_{\mathsf{elem}} \cup \{|\mathsf{elem}| \geq k_0\}$.

$(2 \Rightarrow 1)$. Let $\sim_{\mathsf{elem}}$ be an equivalence relation of P_{elem} for which our method outputs `succeed`. Denote with $\sim_{\mathsf{list}}$ and $\prec_{\mathsf{list}}$ the relations of P_{list} constructed in the list phase, and denote with k_0 the minimal integer computed in the element phase. Next, note that there exists a structure $\mathcal{A}$ satisfying α_{elem} and a T_{int}-structure $\mathcal{B}$ satisfying $\Gamma_{\mathsf{int}} \cup \alpha_{\mathsf{int}}$.

Using $\mathcal{A}$ and $\mathcal{B}$, we define a T_{lint}-structure $\mathcal{C}$ satisfying $\alpha_{\mathsf{elem}} \cup \Gamma_{\mathsf{int}} \cup \Gamma_{\mathsf{list}} \cup \Gamma_{\mathsf{length}}$ by first letting $C_{\mathsf{elem}} = A_{\mathsf{elem}} \cup X$, where X is any infinite set disjoint from A_{elem}. We also let:

$$\begin{aligned} e^{\mathcal{C}} &= e^{\mathcal{A}}, && \text{for all } e \in \mathit{pars}_{\mathsf{elem}}(\Gamma)\,, \\ u^{\mathcal{C}} &= u^{\mathcal{B}}, && \text{for all } u \in \mathit{pars}_{\mathsf{int}}(\Gamma)\,. \end{aligned}$$

In order to define $\mathcal{C}$ over the list-parameters in P_{list}, we fix an injective function $h : (P_{\mathsf{list}} \,/\, \sim_{\mathsf{list}}) \to X$. Note that h exists because P_{list} is finite and X is infinite.

Next, we proceed by induction on the well-founded relation $\prec_{\mathsf{list}}$. Thus, let $x \in P_{\mathsf{list}}$. Then:

– In the **base case**, we let $x^{\mathcal{C}}$ be the unique list of length $u_x^{\mathcal{B}}$ containing only the element $h([x]_{\sim_{\mathsf{list}}})$. In other words, $x^{\mathcal{C}}(i) = h([x]_{\sim_{\mathsf{list}}})$ for $i < u_x^{\mathcal{B}}$, and $x^{\mathcal{C}}(i) = \bot$ for $i \geq u_x^{\mathcal{B}}$.
– In the **inductive case**, fix a list-parameter y such that $x \prec_{\mathsf{list}} y$. Then there exists parameters x', y', e such that $x \sim_{\mathsf{list}} x'$, $y \sim_{\mathsf{list}} y'$, and the literal $x' \approx \mathsf{cons}(e, y')$ is in Γ_{list}. We let $x^{\mathcal{C}} = \mathit{cons}(e^{\mathcal{M}}, (y')^{\mathcal{M}})$.

Note that $\mathcal{C}$ is well-defined over the list-parameters. Furthermore, by construction $\mathcal{C}$ is a T_{lint}-structure satisfying $\alpha_{\mathsf{elem}} \cup \Gamma_{\mathsf{int}} \cup \Gamma_{\mathsf{list}} \cup \Gamma_{\mathsf{length}}$.

It follows that there exists a T-structure $\mathcal{D}$ satisfying $\alpha_{\mathsf{elem}} \cup \Gamma_{\mathsf{int}} \cup \Gamma_{\mathsf{list}} \cup \Gamma_{\mathsf{length}}$ and such that $|D_{\mathsf{elem}}| = k_0$. But then, we can use $\mathcal{D}$ and $\mathcal{A}$ to obtain a T-structure $\mathcal{M}$ satisfying Γ by letting $M_{\mathsf{elem}} = A_{\mathsf{elem}}$ and

$$\begin{aligned}
e^{\mathcal{M}} &= e^{\mathcal{A}}, && \text{for all } e \in \Sigma^{\mathrm{C}}_{\mathsf{elem}} \cup \mathit{pars}_{\mathsf{elem}}(\Gamma)\,,\\
f^{\mathcal{M}} &= f^{\mathcal{A}}, && \text{for all } f \in \Sigma^{\mathrm{F}}_{\mathsf{elem}}\,,\\
p^{\mathcal{M}} &= p^{\mathcal{A}}, && \text{for all } p \in \Sigma^{\mathrm{P}}_{\mathsf{elem}}\,,\\
u^{\mathcal{M}} &= u^{\mathcal{D}}, && \text{for all } u \in \mathit{pars}_{\mathsf{int}}(\Gamma)\,.
\end{aligned}$$

In order to define $\mathcal{M}$ over the list-parameters, fix an injective function $g : D_{\mathsf{elem}} \to A_{\mathsf{elem}}$. For convenience, also let $g(\bot) = \bot$. Note that g exists because $|D_{\mathsf{elem}}| = k_0 \leq |A_{\mathsf{elem}}|$. We let:

$$x^{\mathcal{M}} = g(x^{\mathcal{D}}(i))\,, \qquad \text{for all } x \in \mathit{pars}_{\mathsf{list}}(\Gamma) \text{ and } i \in \mathbb{N}\,.$$

By construction, $\mathcal{M}$ is a T-structure satisfying Γ. ∎

From Proposition 12 and the fact that our combination method is terminating, we obtain the following decidability result.

Theorem 13 (Decidability). *Let T_{elem} be a Σ_{elem}-theory with a decidable ground satisfiability problem. Then the ground satisfiability problem of $\mathit{comb}(T_{\mathsf{elem}}, T_{\mathsf{lint}})$ is decidable.* □

5 Using the combination method

In this Section, we describe how to lift the proposed combination method to efficiently (at least in practice) handle arbitrary Boolean combinations of ground literals. The method is a refinement of the main loop of **haRVey** [6] (cf. Figure 2), a prover based on a combination of Boolean solving and satisfiability checking modulo theories. The idea is to obtain a propositional abstraction φ^a of a formula φ (cf. *abs*) and to enumerate all the propositional assignments (cf. *pick_assign*). If an assignment, refined to a conjunction of first-order literals (cf. *prop2fol*), is found satisfiable modulo the background theory (cf. *check_sat*), then we are entitled to conclude the satisfiability of φ. Otherwise, a new assignment is considered. For efficiency, it is crucial to reduce the number of invocations to *check_sat*. To this end, it is required that *check_sat* returns a conflict set π (which

```
 1: φ := preprocess(φ)
 2: φ^a ← abs(φ)
 3: while φ^a ≠ false do
 4:    Γ^a ← pick_assign(φ^a)
 5:    Γ ← prop2fol(Γ^a)
 6:    (ρ, π) ← check_sat(Γ)
 7:    if ρ = fail then
 8:       φ^a ← φ^a ∧ ¬fol2prop(π)
 9:    else
10:       return succeed
11:    end if
12: end while
```

Fig. 2: haRVey's main loop.

is a subset of the input set of literals) so that all the propositional assignments sharing that set can be eliminated in one shot.

We now give some details of the implementation of the functionalities in Figure 2 which are peculiar to using the combination method in Section 3. In particular, we describe how to satisfy the requirements necessary for the method to work correctly (see beginning of Section 3) and, most importantly, we explain how to compute the $\sim_{\mathsf{list}}$ and $\prec_{\mathsf{list}}$ of Section 3.2.

Function preprocess. A *flat atom* is an atom of the form $p(c_1, \ldots, c_n)$, $c \approx f(c_1, ..., c_m)$, $c_1 \approx c_2$ or $c_1 \approx d$, where p is n-ary predicate symbol ($n \geq 0$), f is an m-ary function symbol ($m > 0$), c_i is an element of **par**, and d is a constant. A *flat literal* is either a flat atom or the negation of a flat atom of one of the two forms $\neg p(c_1, \ldots, c_n)$ or $c_1 \not\approx c_2$. A formula is said to be flattened if all its literals are flat. It is easy to get an equisatisfiable flattened formula from any ground formula by introducing fresh parameters to name subterms.

The preprocessing step also removes all occurrences of car and cdr in the formula using the following equivalences

$$\begin{aligned} e \approx \mathsf{car}(x) &\equiv (x \approx \mathsf{nil} \wedge e \approx \bot_{\mathsf{elem}}) \vee (x \not\approx \mathsf{nil} \wedge (\exists_{\mathsf{list}}\ y)(x \approx \mathsf{cons}(e, y))) \\ x \approx \mathsf{cdr}(y) &\equiv (y \approx \mathsf{nil} \wedge x \approx \bot_{\mathsf{list}}) \vee (y \not\approx \mathsf{nil} \wedge (\exists_{\mathsf{elem}}\ e)(y \approx \mathsf{cons}(e, x))) \end{aligned}$$

For instance, $\varphi[a \approx \mathsf{car}(x)]$ is equisatisfiable to $\varphi[a \approx e] \wedge e \approx \mathsf{car}(x)$. In this last formula, the atom $e \approx \mathsf{car}(x)$ has always positive polarity. In a later step, it can be replaced by $(x \approx \mathsf{nil} \wedge e \approx \bot_{\mathsf{elem}}) \vee (x \not\approx \mathsf{nil} \wedge (\exists_{\mathsf{list}}\ y)(x \approx \mathsf{cons}(e, y)))$ and since the polarity is positive, the existential quantifier can be Skolemized by simply introducing a fresh parameter. Exhaustively applying this transformation gives a new ground formula, without car and cdr.

Finally, and still by introducing fresh parameters, functions cons and length are made to appear only in unit clauses of the form $\mathsf{cons}(e, x) \approx y$ or $\mathsf{length}(x) \approx u$. For instance formula $\varphi[\mathsf{cons}(e, x) \not\approx y]$ is replaced by $\varphi[y' \not\approx y] \wedge y' \approx \mathsf{cons}(e, x)$.

Function pick_assign. The function *pick_assign* is implemented by the Boolean solver and returns a propositional assignment satisfying φ^a. It is easy to tune the solver to make *pick_assign* return a propositional assignment Γ^a such that $\mathit{prop2fol}(\Gamma^a)$ contains the literals representing the fact that each list parameter is equal to nil or not.

Function check_sat. First of all, we notice that, thanks to *preprocess*, the function *pick_assign* returns a set Γ of literals which can be put in separate form satisfying conditions (a)–(e) at the beginning of Section 3 by simply partitioning the literals.

Our combination method uses decision procedures for the quantifier-free fragment of arithmetic and for the theory of acyclic lists. While we use a decision procedure for the first theory as a black box, we require the decision procedure for the theory of acyclic lists to be able to return $\sim_{\mathsf{list}}$ and $\prec_{\mathsf{list}}$. For this reason, we detail below how to do this.

Reasoning about acyclic lists

We introduce a graph structure encapsulating all constraints on the T_{list}-models of a set of equalities of the form $x \approx_{\mathsf{list}} y$, $e \approx_{\mathsf{elem}} e'$, $x \approx \mathsf{cons}(e, y)$. This structure can be easily computed, and the required relations can be immediately deduced from it. Furthermore, it may be used in order to guide the guessing in Section 3.1.

From now on, if not otherwise specified, nil is treated as any other parameter. An equality $x \approx \mathsf{nil}$ can thus be seen as an equality between two different list parameters. Given finite sets of list and element parameters, a list-graph is a tuple $\langle V_{\mathsf{list}}, V_{\mathsf{elem}}, s_{\mathsf{list}}, s_{\mathsf{elem}} \rangle$ with

- V_{list} (V_{elem}) is a partition of list (resp. element) parameters. It is the set of list (resp. element) nodes. Parameters in a node are *labels* for that node;
- s_{list} (s_{elem}) is a function from V_{list} to subsets of V_{list} (resp. V_{elem}). Given a list node u, $s_{\mathsf{list}}(u)$ ($s_{\mathsf{elem}}(u)$) is the set of list (resp. element) successors of u.

A T_{list}-structure $\mathcal{A}$ agrees with a list-graph if the following conditions are met:

- if x and y label the same node then $\mathcal{A} \models x \approx y$, where x and y are both element parameters or both list parameters;
- if y labels the list successor of x then $\mathcal{A} \models \exists e\ x \approx \mathsf{cons}(e, y)$;
- if e labels the element successor of x then $\mathcal{A} \models \exists y\ x \approx \mathsf{cons}(e, y)$.

Assume L is a T_{list}-satisfiable set of equalities of the form $x \approx_{\mathsf{list}} y$, $e \approx_{\mathsf{elem}} e'$, $x \approx \mathsf{cons}(e, y)$. Then there is a list-graph G such that, for every T_{list}-structure $\mathcal{A}$, $\mathcal{A}$ agrees with G if and only if $\mathcal{A}$ is a model of L. Indeed, the following graph verifies this property:

- x and y label the same node if and only if $L \models_{\mathsf{list}} x \approx y$,[6] where x and y are both element parameters or both list parameters;

[6] $\models_{\mathsf{list}}$ denotes logical consequence in the theory of lists. That is $L \models_{\mathsf{list}} x \approx y$ if every T_{list}-model of L is a model of $x \approx y$.

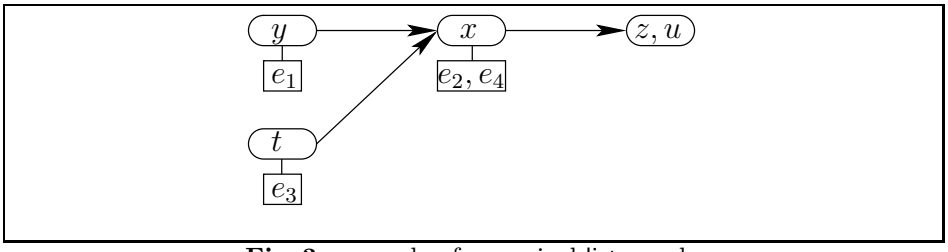

Fig. 3: example of canonical list-graph

- y labels the list successor of x if and only if $L \models_{\mathsf{list}} \exists e\ x \approx \mathsf{cons}(e, y)$;
- e labels the element successor of x if and only if $L \models_{\mathsf{list}} \exists y\ x \approx \mathsf{cons}(e, y)$.

This graph is unique. It is such that, for each $v \in V_{\mathsf{list}}$, $s_{\mathsf{list}}(v)$ and $s_{\mathsf{elem}}(v)$ are either a singleton or the empty set. In other words, every list node has at most one list successor, and one element successor. In fact, it can be showed that every node has two or zero successor, since the cdr and car functions are not explicitly used in the set of equalities. If nil labels a list-node, then this node has no list successors. It is acyclic in the sense that s_{list} is acyclic. Finally, for each $u, v \in V_{\mathsf{list}}$, if $s_{\mathsf{list}}(u) = s_{\mathsf{list}}(v)$, $s_{\mathsf{list}}(u) \neq \emptyset$, $s_{\mathsf{elem}}(u) = s_{\mathsf{elem}}(v)$, and $s_{\mathsf{elem}}(u) \neq \emptyset$, then $u = v$. In other words, two different list nodes must not have the same list and element successors.

This graph will thus be called the *canonical* list-graph for a set of equalities. For instance, the canonical list-graph for the set of equalities

$$y \approx \mathsf{cons}(e_1, x), x \approx \mathsf{cons}(e_2, z), x \approx \mathsf{cons}(e_4, u), t \approx \mathsf{cons}(e_3, x)$$

is given in Figure 3.

Given the canonical list-graph for a set of equalities, we have that $x \sim_{\mathsf{list}} y$ is true if and only if *x and y both label the same list node and $\prec_{\mathsf{list}}$ is the transitive closure of the list successor relation.*

Computing canonical list-graphs

To compute the canonical graph for a set of equalities, three transformations on list-graphs are necessary:

- a congruence step replaces two lists nodes u and v such that $s_{\mathsf{list}}(u) = s_{\mathsf{list}}(v)$ and $s_{\mathsf{elem}}(u) = s_{\mathsf{elem}}(v)$ by a unique node $u \cup v$.[7] The new node inherits all successors of the nodes it replaces. All list nodes which had u or v as list successor are made to have $u \cup v$ as list successor.
- a list unification step (Unify-cdr) replaces two list successors u and v of one node t by a unique node $u \cup v$. The new node inherits all successors of the nodes it replaces. All list nodes which had u or v as list successor are made to have $u \cup v$ as list successor.

[7] Remember u and v are disjoint sets of list parameters.

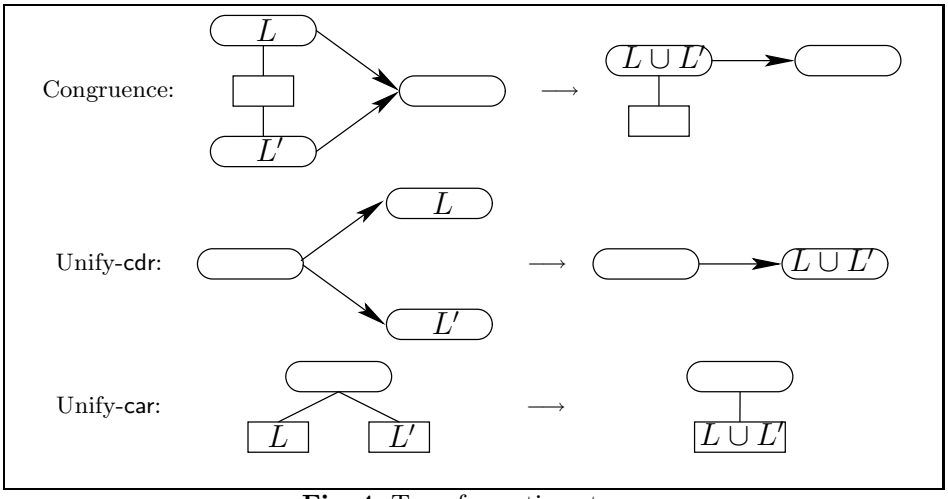

Fig. 4: Transformation steps

– an element unification step (Unify-car) replaces two element successors u and v of one node t by a unique node $u \cup v$. All list nodes which had u or v as element successor are made to have $u \cup v$ as list successor.

These transformations are depicted in Figure 4.

Let L be a set of equalities of the form $x \approx_{\mathsf{list}} y$, $e \approx_{\mathsf{elem}} e'$, $x \approx \mathsf{cons}(e, y)$. To build the canonical graph for this set, the first operation is to compute the reflexive, symmetric and transitive closure of all equalities between parameters in the set L. Second, for every equality $\mathsf{cons}(e, x) \approx y$, the nodes labeled by x and e are made list and element successors of the node labeled by y. Third, the graph is unified, beginning with nodes without parent, finishing with those without successor, using unification steps (beginning with all element unification steps). Last, the congruence rule is applied, from the nodes without successors, to the nodes without parents. In presence of nil, a postprocessing ensures that the node it labels has no successor.

If the graph happens to be cyclic, or if nil happens to have a successor, the procedure fails. In that case the initial set of equalities is unsatisfiable. A careful implementation of this procedure is linear in time [13].

The obtained graph (after a finite number of transformation steps) is indeed the canonical graph: every T_{list}-structure $\mathcal{A}$ agreeing with a graph G also agrees with the graph obtained from G by a transformation step. That ensures that every model of L agrees with the final graph. To show that every T_{list}-structure agreeing with the graph is also a model for L, it suffices to show that every equality of L is trivially satisfied by any structure agreeing with the graph.

There is a T_{list}-structure agreeing with a canonical list-graph, such that every node is assigned to a different element or list. As a consequence, satisfiability checking of a set of literals in T_{list} can be simply implemented by building the

canonical list-graph for all equalities in the set, and check afterwards if no inequality has both members labeling the same node.

Two final remarks are in order. First, the list-graph may be build before guessing an arrangement of the element parameters, and may be used to guide this guessing. Indeed it is not necessary to consider an α_{elem} implying that two parameters labeling the same node in the list-graph are different. Second, for the algorithm in Figure 2 to be efficient, it is required also that *check_sat* returns a small (minimal, if possible) conflict set π out of the input set of literals. For instance, the decision procedure for acyclic lists should produce small unsatisfiable subsets of the input set of literals, or be able to give the equations necessary to deduce a given equality from a satisfiable set. We believe this is possible by adapting the method developed for congruence closure in [5].

6 Conclusion

We presented a combination method that is able to combine a many-sorted theory T_{lint} modeling lists of elements in the presence of the length operator with a theory T_{elem} modeling the elements.

Our method works regardless of whether the theory of the elements is stably infinite or not. We were able to relax the stable infiniteness requirement by employing the following basic ideas:

- guess an arrangement larger than the one computed by Nelson and Oppen;
- compute a certain minimal cardinality k_0, so that we can ensure that the domain of the elements must have at least k_0 elements.

We plan to implement the proposed method in **haRVey**. In particular, we will investigate extending the procedure for acyclic lists to compute minimal conflict sets.

Acknowledgments

We are grateful to Christophe Ringeissen for insightful discussions on the problem of combining non-stably infinite theories.

References

1. A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, 183(2):140–164, 2003.
2. S. Berezin, V. Ganesh, and D. L. Dill. An Online Proof-Producing Decision Procedure for Mixed-Integer Linear Arithmetic. In *Proceedings of TACAS'03*, volume 2619 of *LNCS*, Warshaw, Poland, April 2003.
3. N. S. Bjørner. *Integrating Decision Procedures for Temporal Verification*. PhD thesis, Stanford University, 1998.
4. R. S. Boyer and J. S. Moore. *A Computational Logic*. ACM Monograph SERIES, 1979.

5. L. de Moura, H. Rueß, and N. Shankar. Justifying equality. In *PDPAR*, 2004.
6. D. Déharbe and S. Ranise. Light-Weight Theorem Proving for Debugging and Verifying Units of Code. In *Proc. of the International Conference on Software Engineering and Formal Methods (SEFM03)*. IEEE Computer Society Press, 2003.
7. J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: integrated canonizer and solver. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification (CAV)*, volume 2102 of *LNCS*, pages 246–249. Springer-Verlag, 2001.
8. P. Fontaine and P. Gribomont. Combining non-stably infinite, non-first order theories. In S. Ranise and C. Tinelli, editors, *Pragmatics of Decision Procedures in Automated Reasoning*, 2004.
9. H. Ganzinger. Shostak light. In A. Voronkov, editor, *Automated Deduction – CADE-18*, volume 2392 of *LNCS*, pages 332–346. Springer, 2002.
10. D. Kapur and X. Nie. Reasoning about Numbers in Tecton. In *Proc.* 8^{th} *Inl. Symp. Methodologies for Intelligent Systems*, pages 57–70, 1994.
11. T. F. Melham. Automating Recursive Type Definitions in Higher Order Logic. In *Current Trends in Hardware Verification and Theorem Proving*, LNCS, pages 341–386. Sprigner-Verlag, 1989.
12. G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the Association for Computing Machinery*, 27(2):356–364, 1980.
13. D. C. Oppen. Reasoning about recursively defined data structures. *Journal of the ACM*, 27(3):403–411, 1980.
14. S. Owre and N. Shankar. Abstract Datatypes in PVS. Technical Report CSL-93-9R, SRI International, 1997.
15. L. C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In A. Bundy, editor, *Automated Deduction — CADE-12*, LNAI 814, pages 148–161. Springer, 1994. 12th international conference.
16. W. Pugh. The omega test: a fast integer programming algorithm for dependence analysis. *Supercomputing*, pages 4–13, 1991.
17. R. E. Shostak. Deciding combination of theories. *Journal of the Association for Computing Machinery*, 31(1):1–12, 1984.
18. A. Stump, C. W. Barrett, and D. L. Dill. CVC: a cooperating validity checker. In E. Brinksma and K. G. Larsen, editors, *Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 500–504. Springer, 2002.
19. C. Tinelli and C. G. Zarba. Combining non-stably infinite theories. *Journal of Automated Reasoning*, 2004. To appear.
20. P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In S. Graf and M. I. Schwartzbach, editors, *TACAS*, volume 1785 of *LNCS*, pages 1–19, Berlin, Mar. 2000. Springer-Verlag.
21. C. G. Zarba. Combining multisets with integers. In A. Voronkov, editor, *Automated Deduction – CADE-18*, volume 2392 of *LNCS*, pages 363–376. Springer, 2002.
22. C. G. Zarba. Combining sets with integers. In A. Armando, editor, *Frontiers of Combining Systems*, volume 2309 of *LNCS*, pages 103–116. Springer, 2002.
23. C. G. Zarba. Combining sets with elements. In N. Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 762–782. Springer, 2004.
24. T. Zhang, H. B. Sipma, and Z. Manna. Decision procedures for recursive data structures with integer constraints. In D. A. Basin and M. Rusinowitch, editors, *Automated Reasoning*, volume 3097 of *LNCS*, pages 152–167. Springer, 2004.