



Efficient Linear Model Building and Clause Evaluation

Simon Schwarz^{1,2}, Christoph Weidenbach¹, and Fabian Wobito¹

¹ Max Planck Institute for Informatics, Saarbrücken, Germany
{sschwarz,weidenbach,fwobito}@mpi-inf.mpg.de

² Graduate School of Computer Science, Saarland Informatics Campus, Saarbrücken, Germany

Abstract

We describe our implementation of the linear model building calculus first presented in [5]: Given a finite consistent set of ground literals, a so-called trail, we want to extend it to a complete Herbrand-style interpretation represented by linear literals. Moreover, we give an algorithm to evaluate arbitrary clauses in the resulting linear model. We provide concrete data structures for efficiently implementing model building and clause evaluation. The resulting model building procedure runs in time linear in the size of the trail for a fixed signature. Even though checking whether a clause is false in a linear model is worst-case NP-complete, we provide a lazy backtracking algorithm that avoids materializing intermediate substitutions and is efficient in practice.

Our proposed algorithms are implemented as part of the SPASS(SCL) solver. Our experimental evaluation on all equality-free first-order TPTP problems shows that model building scales linearly in practice and, together with clause evaluation, accounts for less than one percent of the solver runtime on average on nontrivial runs. The generated models establish satisfiability for roughly a third of the satisfiable TPTP problems considered. If a generated interpretation is not a model of the input clauses, clause evaluation provides a concrete counterexample for extending the trail.

Keywords: model building; first-order logic; SCL(FOL); SPASS(SCL)

1 Introduction

Simple Clause Learning for first-order logic (SCL(FOL)) [10, 7] explores first-order clause sets with respect to ground partial models, called trails. Similar to CDCL, such a trail is a consistent sequence of ground literals obtained by decisions and propagations. The size of the trail is always finitely bounded at every point in time. Thus, even if all considered ground instances are satisfied, the trail is not an overall model of the input clause set, in general.

SCL(FOL) therefore regularly reaches so-called stuck states. For a fixed bound on the considered ground literals, no conflict is found and no further inference is possible, but this only means that the trail is a model for the bounded set of ground instances. There are several ways to proceed. The calculus may increase the bound and continue, which is sufficient for refutational completeness under suitable fairness assumptions. It may also search for different partial models at the same bound [6]. Alternatively, it may first explore the information already contained in the trail. If the trail can be extended to a complete first-order interpretation, then

this interpretation can be checked against the original clauses. If all clauses are satisfied, a model has been found. If not, the false clause instance gives a concrete reason for increasing the bound, prevents the re-generation of the computed model in the future, and may guide the next search step.

Bromberger et al. [5] recently introduced a model generation procedure for precisely this situation. Given a finite consistent set of ground literals, their calculus generates a complete Herbrand-style interpretation represented by linear first-order literals. The construction starts from a universal relation for each predicate and refines it according to the term structure of positive and negative trail literals until every remaining piece can be assigned a polarity. This paper is concerned with turning this abstract model generation procedure into an efficient implementation inside the SPASS workbench [24], including the subsequent task of evaluating clauses in the generated interpretation.

Let us illustrate the model construction on a small example. Consider the signature consisting of a binary predicate P , a unary function f , and a constant a , and let the trail be

$$\Gamma = \{\neg P(f(f(a)), f(a)), \neg P(f(a), a), P(f(f(f(a))), f(a)), \neg P(f(f(a)), a)\}.$$

Starting from the atom $P(x, y)$, the model generation procedure refines variable positions where the positive and negative instances in Γ force different behaviour. One possible complete linear interpretation obtained in this way is

$$\mathcal{M} = \{\neg P(f(x), a), \neg P(f(f(a)), f(x)), P(f(f(f(x))), f(y)), \\ P(a, x), P(f(a), f(x))\}.$$

This finite set of linear literals assigns a truth value to every ground atom over the signature and satisfies all literals in Γ . For example, atoms of the form $P(f(t), a)$ are negative, atoms of the form $P(f(f(f(s))), f(t))$ are positive, and atoms not constrained by the trail are completed by the construction. The representation is finite, but the interpretation is infinite.

Once such an interpretation has been generated, the next question is whether it satisfies the original clause set. This is the algorithmically more delicate part. For example, consider the clause

$$C = \neg P(x, x) \vee P(f(x), f(a))$$

with respect to the interpretation $\mathcal{M}$ above. To falsify C , both literals must be false under the same grounding substitution. Searching the model shows that the relevant case is $\{x \mapsto f(a)\}$, yielding the false instance

$$\neg P(f(a), f(a)) \vee P(f(f(a)), f(a)).$$

In general, clause evaluation has to search through possible instantiations of the clause variables. The search may branch, and the substitutions constructed along the way may contain fresh helper variables. Thus, a direct implementation that repeatedly materializes substituted atoms or collects all intermediate substitutions would introduce avoidable overhead.

Our contribution is an implementation-oriented account of linear model building and clause evaluation for SCL(FOL). First, we introduce *refinement trees*, a compact representation of the refinement structure generated by the model calculus. Instead of storing every refined atom explicitly, a refinement tree stores only the selected refinement variables, the function symbols on edges, and the polarity of leaves. The represented linear literals can be reconstructed from the path to a leaf, while traversal of the tree directly implements atom evaluation. Refinement trees can be viewed as specializations of classical first-order indexing structures where terms and

atoms are also stored in tree-like structures for efficient access [14, 11, 17, 21]. Due to the fact that refinement trees only store linear atoms and terms, they can recover variable information implicitly and therefore just store the function symbol skeleton.

Second, we introduce *subterm views*, a lightweight data structure for model construction and traversal. During refinement, only the subterms at currently relevant variables have to be inspected. A subterm view keeps these subterms in a queue-like array and supports advancing over variables that do not distinguish the current matching trail literals, as well as unfolding the front subterm when descending in the refinement tree. This yields a direct divide-and-conquer implementation of model construction. In particular, the model for a trail can be built in time linear in the size of the input trail, up to the unavoidable dependence on the signature.

Third, we present clause evaluation as lazy backtracking over refinement-tree traversals. For each literal of a clause, the traversal computes the possible extensions of the current substitution that force the required truth value. Substitutions are applied lazily: bindings are inspected only when the current tree node needs the corresponding top symbol, and both substitution state and subterm views are restored in-place during backtracking. This realizes the mathematical set of possible instantiations without materializing it.

Finally, we implemented these data structures and algorithms as part of the SPASS(SCL) solver. We evaluate the implementation on all equality-free first-order problems from the TPTP library [23]. The experiments show that model construction behaves linearly in the trail size in practice and that model construction and clause evaluation usually account for only a small fraction of the total solver runtime. Thus, the generated linear models are not merely a theoretical completion device for stuck states, but can be used inside the prover loop with low overhead.

The paper is structured as follows: Section 3 presents efficient linear model building, including refinement trees and subterm views. Section 4 explains clause evaluation by lazy backtracking. Section 5 discusses the integration into SPASS(SCL), and Section 6 reports experimental results. Finally, Section 7 concludes.

2 Preliminaries

We assume a first-order language without equality over a finite signature $\Sigma = (\Omega, \Pi)$ of function symbols and predicates, respectively. All signature symbols have a fixed arity, where constants are nullary function symbols. We write $f/1$ or $R/2$ for a function symbol of arity 1 or a predicate symbol of arity 2, respectively. Let N denote a clause set; C, D denote clauses; L, K, H denote literals; A, B denote atoms; P, Q, R denote predicates; t, s terms; f, g, h function symbols; a, b, c constants; and x, y, z variables. Atoms, literals, clauses and clause sets are considered as usual, where in particular clauses are identified both with their disjunction and the multiset of their literals. In this work, we sometimes annotate an atom's explicit polarity as $+A$ or $\neg A$. The complement of a literal is denoted by the function comp. The atom of a literal L is denoted by $\text{atom}(L)$, where $\text{atom}(A) = A$ and $\text{atom}(\neg A) = A$. Semantic entailment $\models$ is defined as usual where variables in clauses are assumed to be universally quantified. Substitutions σ, τ are total mappings from variables to terms, where $\text{dom}(\sigma) := \{x \mid x\sigma \neq x\}$ is finite and $\text{codom}(\sigma) := \{t \mid x\sigma = t, x \in \text{dom}(\sigma)\}$. Their application is extended to literals, clauses, and sets of such objects in the usual way. In particular, we refer to a clause $C\sigma$ as a *clause instance* of C and call a literal $L\sigma$ an instance of L . A literal L is an *atom instance* of a literal K if $\text{atom}(K)\sigma = \text{atom}(L)$ for some σ . The composition $\sigma\tau$ first applies σ and then τ , i.e., $t(\sigma\tau) = (t\sigma)\tau$. The function $\text{vars}(Z)$ returns the set of variables in a term/atom/literal/clause/clause set Z . A term, atom, clause, or a set of these objects is

ground if it does not contain any variable. A substitution σ is *ground* if $\text{codom}(\sigma)$ is ground. A substitution σ is *grounding* for a term t , literal L , clause C if $t\sigma$, $L\sigma$, $C\sigma$ are ground, respectively. The function mgu denotes the *most general unifier* of two terms, atoms, literals. We assume that any mgu of two terms or literals does not introduce any fresh variables and is idempotent.

A term, atom, or literal is *linear* if every variable occurs at most once. Two literals are *inconsistent* if they have different polarity and unifiable atoms. A set of literals (often also called a *trail*) is *consistent* if no pair of literals in the set is inconsistent. A *literal interpretation* is a finite consistent set of literals. It assigns a ground atom positive polarity if the atom is an instance of a positive literal in the set, negative polarity if it is an instance of a negative literal in the set, and leaves it undefined otherwise. A literal interpretation is *complete* if it leaves no ground atom undefined. A complete literal interpretation consisting of linear literals is called a *linear model*.

The size of a variable is 1, and the size of a term or atom is one plus the sum of the sizes of its immediate subterms. The size of a literal is the size of its atom, and the size of a set of literals is the sum of the sizes of its literals.

2.1 The Linear Model Generation Calculus

The model generation calculus we implement was first published in [5]. Here, we recall only its formal rules; Section 3 revisits the construction more intuitively and illustrates it with examples while developing an efficient implementation. The calculus operates on states $(\Gamma; \Delta; \mathcal{I}; \mathcal{M})$, where Γ and Δ are consistent sets of ground literals, $\mathcal{M}$ is a set of linear literals defining a partial interpretation such that $\mathcal{M} \models L$ for each $L \in \Delta$ and $\mathcal{M}$ does not conflict with Γ , and $\mathcal{I}$ is a set of linear atoms such that $\mathcal{I} \cup \mathcal{M}$ represents a complete literal interpretation. The rules move processed literals from Γ to Δ and processed atoms from $\mathcal{I}$ to $\mathcal{M}$. The calculus builds an interpretation for a single predicate P , and Γ is assumed to contain only P -literals. For several predicates, a separate run is performed on the respective literals of each predicate. Commas within a component denote disjoint union or disjoint addition of an element. For a position p in an atom A , we write $A|_p$ for the subterm of A at p .

$$\begin{array}{l} \textbf{Refine} \quad (\Gamma; \Delta; \mathcal{I}, P(t_1, \dots, t_n); \mathcal{M}) \\ \Rightarrow_{\text{mod}} (\Gamma; \Delta; \mathcal{I}, \bigcup_{f_i/k_i \in \Omega} \{P(t_1, \dots, t_n)\{x \mapsto f_i(y_{i,1}, \dots, y_{i,k_i})\}\}; \mathcal{M}) \end{array}$$

provided $P(t_1, \dots, t_n)|_p = x$, f_i are all function symbols (including constants) from Ω and k_i denotes their respective arity, all variables in $f_i(y_{i,1}, \dots, y_{i,k_i})$ are fresh, different variables, and p is a depth-minimal variable position in $P(t_1, \dots, t_n)$ for which there exist literals $P(l_1, \dots, l_n)$ and $\neg P(r_1, \dots, r_n)$ in Γ such that both are atom instances of $P(t_1, \dots, t_n)$, and the subterms $P(l_1, \dots, l_n)|_p$ and $P(r_1, \dots, r_n)|_p$ are not unifiable

$$\begin{array}{l} \textbf{Clean} \quad (\Gamma; \Delta; \mathcal{I}, P(t_1, \dots, t_n); \mathcal{M}) \\ \Rightarrow_{\text{mod}} (\Gamma; \Delta; \mathcal{I}; \mathcal{M}, P(t_1, \dots, t_n)) \end{array}$$

provided $P(t_1, \dots, t_n)$ has no atom instance in Γ

$$\begin{array}{l} \textbf{Solve} \quad (\Gamma, \Gamma'; \Delta; \mathcal{I}, P(t_1, \dots, t_n); \mathcal{M}) \\ \Rightarrow_{\text{mod}} (\Gamma; \Delta, \Gamma'; \mathcal{I}; \mathcal{M}, \#P(t_1, \dots, t_n)) \end{array}$$

provided $\Gamma' \neq \emptyset$ and Γ' consists only of positive literal instances of $P(t_1, \dots, t_n)$ or only of negative literal instances; Γ contains no literal instances of $P(t_1, \dots, t_n)$; $\# = \neg$ if the literals in Γ' are negative and $\#$ is empty otherwise

For a predicate P/n , the initial state is $(\Gamma; \emptyset; \{P(x_1, \dots, x_n)\}; \emptyset)$. The abstract calculus terminates in time $O(\text{size}(\Gamma)^2)$ in a state $(\emptyset; \Delta; \emptyset; \mathcal{M})$, where $\mathcal{M}$ is a complete linear model for P satisfying the initial Γ [5]. In the next section, we turn this abstract calculus into a concrete algorithm that, for a fixed signature, improves the quadratic runtime bound to a linear one.

3 Efficient Linear Model Building

We now turn the linear model generation calculus recalled in Section 2.1 into an efficient procedure. Given a consistent finite set Γ of ground literals over a predicate P , it produces a linear model for P that respects the structure of Γ . We introduce refinement trees to record the successive applications of *Refine* and the polarities assigned at the leaves. We then introduce subterm views and show how both data structures work together in a model-building algorithm that runs in time linear in the size of Γ .

Without loss of generality, we focus on a single predicate P throughout this section. If several predicates occur in Γ , the same construction is simply carried out independently for each of them. For a linear atom A , we call $\Gamma_A := \{L \in \Gamma \mid \text{atom}(L) \text{ is an instance of } A\}$ its set of *matching trail literals*.

3.1 Refinement Trees

Recall what *Refine* does in the abstract calculus. Given a linear atom that still has both positive and negative matching trail literals, *Refine* picks a variable at which matching positive and negative trail atoms differ. It then replaces the atom by all depth-1 instances for that variable: one new atom for each function symbol of the signature. For a small example, take the trail $\Gamma = \{P(a), \neg P(f(b))\}$ over constants a, b and a unary function f , and start from the atom $P(x)$. Both signs occur among the matching trail literals and their terms differ at x , so *Refine* applies at x and replaces $P(x)$ by the three atoms $P(a)$, $P(b)$, and $P(f(x))$. The trail literal $P(a)$ is an instance of the first, $\neg P(f(b))$ is an instance of the third, and no trail literal is an instance of $P(b)$. Thus, an overall linear model can be obtained as $\{P(a), P(b), \neg P(f(x))\}$. The *Clean* rule of the calculus assigns the empty branch $P(b)$ the default positive polarity.

We record every such *Refine* step as a parent node together with one child per resulting atom: the parent is the atom that was refined, and each child is one of the depth-1 instances. Repeating this construction for every *Refine* step yields a tree, which we call a *refinement tree*.

An internal node of the tree stands for a linear atom that still has both positive and negative matching trail literals in Γ . As long as both signs occur, the atom cannot yet be assigned a polarity. The children of such a node are the depth-1 instances produced by *Refine* at the chosen variable. A leaf is finished when no matching trail literal remains or when all matching trail literals have the same sign. In the former case it receives the default positive polarity; in the latter case it receives their common polarity as defined by the *Solve* rule of the abstract calculus. Each finished leaf therefore represents one linear literal of the final interpretation.

Figure 1 shows the refinement tree for the trail from the introduction. We start from $P(x, y)$. At a suitable variable where the matching trail literals disagree, the tree splits into a child for the constant a and a child for the function symbol f . Along the f -branch we keep refining in the same way, until every remaining set of matching trail literals either has a uniform sign or is empty. The finished leaves then represent the linear literals of the interpretation $\mathcal{M}$ from the introduction, up to variable renaming.

Formally, every node u represents a linear atom, which we denote by $\text{atom}(u)$. The root represents the starting atom $P(x_1, \dots, x_m)$. If an internal node refines a variable x and has a

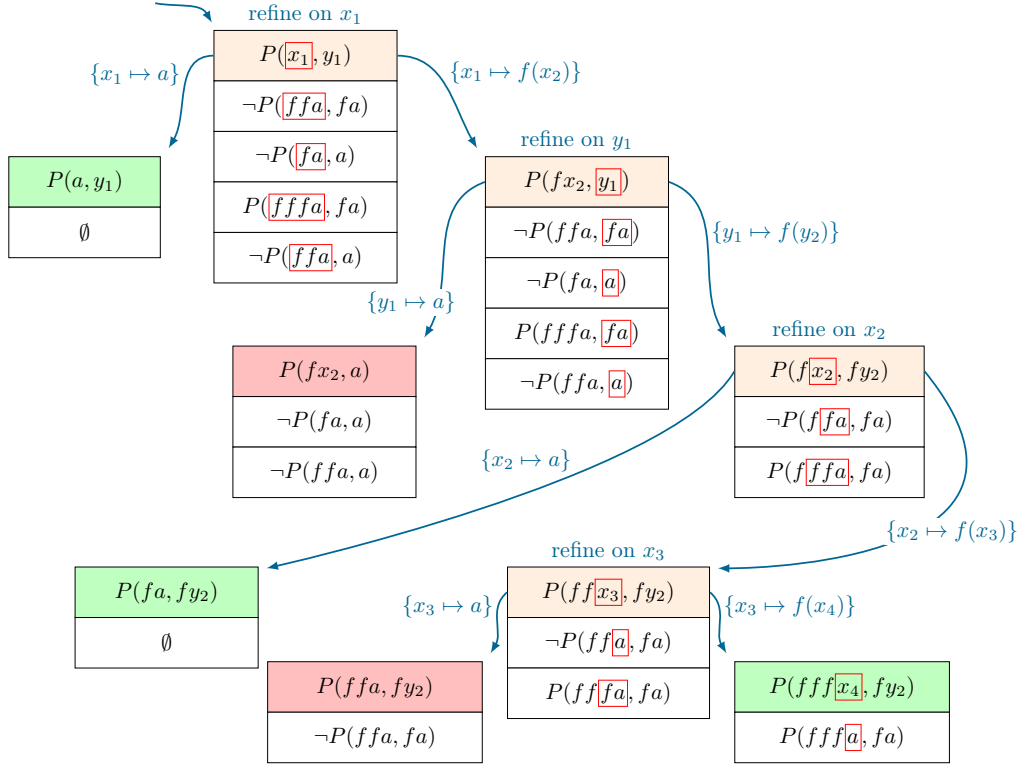


Figure 1: Refinement tree for the trail from the introduction. Each node shows the recovered atom (top) and the matching trail literals (below). We write applications of f in short form, e.g. $fffa$ for $f(f(a))$. Blue labels mark Refine steps. Finished leaves are colour-coded by polarity (green positive, red negative). Their literals form the interpretation $\mathcal{M}$ from the introduction, up to variable renaming.

child corresponding to a function symbol $f/n \in \Omega$, then the child represents the atom obtained by applying the substitution $\{x \mapsto f(y_1, \dots, y_n)\}$ with fresh variables $y_1, \dots, y_n$. In addition, every node carries a polarity status. Internal nodes are unfinished, while a finished leaf has either positive or negative polarity status. All finished leaves together with their polarity form the complete linear interpretation for P .

In an implementation we do not need to store the full atom at every node. Instead, each edge is labeled by the corresponding refining substitution. The atom can then be recovered from the root predicate by following a path and applying the recorded substitutions along it. In particular, the domain of an edge substitution records which variable is refined.

3.2 Subterm Views

To build the overall linear model, the *Refine* rule has to be applied exhaustively until every leaf can be assigned a polarity. At each node of the refinement tree, we therefore need to determine which rule applies. If the node has no matching trail literals, *Clean* applies. If all matching trail literals have the same polarity, *Solve* applies and no further refinement is necessary. Otherwise, matching trail literals of both polarities remain, and we refine at a variable at which their subterms differ. Since this check is performed at every unresolved node, we need an efficient

way to find this variable.

For every matching trail literal, a *subterm view* is a standard queue of references to subterms of the literal's atom. Although the queue contains subterms, its entries correspond, in order, to the unexamined variables of the current linear atom. For a trail literal L with $\text{atom}(L) = P(t_1, \dots, t_m)$, its initial subterm view stores the polarity of L and has the queue $[t_1, \dots, t_m]$. Whenever a variable is refined, the subterms corresponding to its freshly introduced argument variables are appended after the entries corresponding to the remaining variables. Thus, the queue determines the fixed order in which the construction considers variables.

There are two operations on a subterm view. First, *advance* removes its front entry (i.e. pops an element from the queue). If the front subterms of all matching trail literals are identical, the corresponding variable cannot distinguish their signs, so all views advance.

Second, *unfold* follows a refinement step. Suppose the current atom contains a variable x , and a matching trail atom instantiates x with $f(t_1, \dots, t_n)$. Applying *Refine* at x uses the substitution $\{x \mapsto f(y_1, \dots, y_n)\}$. The refined atom then contains the fresh variables $y_1, \dots, y_n$ in place of the arguments of f . *Unfold* therefore removes the front $f(t_1, \dots, t_n)$ and appends the sequence $t_1, \dots, t_n$ to the view. The appended entries correspond to the new variables $y_1, \dots, y_n$. In particular, entries corresponding to previously existing variables remain before those corresponding to newly introduced variables. This append-to-back discipline simulates a fixed breadth-first variable order.

Consider the current atom $P(x, y, z)$ and the matching trail literals

$$\Gamma_P = \{P(f(a), g(a), f(f(a))), \neg P(f(a), g(a), f(f(b)))\}.$$

The node cannot be solved because the matching literals have different polarities. The subterms for the first two variables are identical in both literals. Only z distinguishes the two polarities, with subterms $f(f(a))$ and $f(f(b))$. The corresponding subterm views are

$$[f(a), g(a), f(f(a))] \quad \text{and} \quad [f(a), g(a), f(f(b))].$$

To find a suitable variable, we advance the views while their front subterms are equal. The first two advances discard the entries for x and y ; the remaining fronts differ, so the next refinement is performed on z .

For the running example, the current atom contains z , and the two front subterms are $f(f(a))$ and $f(f(b))$. Applying the substitution $\{z \mapsto f(w)\}$ replaces z by $f(w)$ in the refined atom. Unfolding therefore changes the two views to $[f(a)]$ and $[f(b)]$, whose entries correspond to the new variable w . The construction can then continue below the outer f without rebuilding the complete atoms.

We implement a subterm view as a contiguous array with a front index and an append-only back index. Terms are shared, so equality of two front subterms is a constant-time pointer comparison. Each call to *advance* or *unfold* removes one entry. *Unfold* additionally appends precisely the immediate arguments of the removed term. Hence every occurrence in the input term enters and leaves the view at most once. The total work for a subterm view is therefore linear in the size of its trail literal as already processed subterms are neither copied nor traversed again. We also define the size of a subterm view q corresponding to a matching trail literal L to be $\text{size}(q) := \text{size}(L)$. The size of a set of subterm views is the sum of the sizes of its elements.

3.3 The Model Building Algorithm

We now build the refinement tree recursively. Suppose that we have reached an unfinished leaf u . Let $\mathcal{Q}$ be the set of subterm views for the matching trail literals of $\text{atom}(u)$. Each view

retains the polarity of its literal. The views are synchronized: their fronts correspond to the same current variable of $\text{atom}(u)$. Advancing all views skips this variable. Unfolding the views in one branch follows the corresponding refinement and makes its fresh argument variables available later in the queue.

Algorithm 1 directly implements the three model-generation rules. If all fronts have been advanced, inconsistent trail literals have the same atom but opposite polarities. Otherwise, all remaining fronts are ground terms and can be partitioned by their top function symbols.

Algorithm 1: BuildModel($\mathcal{T}, u, \mathcal{Q}$)

Input: A refinement tree $\mathcal{T}$ with unfinished leaf u and synchronized subterm views $\mathcal{Q}$
Effect: Constructs the model for instances of $\text{atom}(u)$, or fails

```

1 if  $\mathcal{Q}$  is empty then                                     // applied rule: Clean
2   | let  $\text{status}(u) \leftarrow \text{finished}_+$ 
3 else if all views in  $\mathcal{Q}$  have the same polarity  $s \in \{+, \neg\}$  then // applied rule: Solve
4   | let  $\text{status}(u) \leftarrow \text{finished}_s$ 
5 else                                                       // applied rule: Refine
6   | while all front subterms in  $\mathcal{Q}$  exist and are equal do
7     | advance all subterm views in  $\mathcal{Q}$ 
8   | end
9   | if some subterm view in  $\mathcal{Q}$  is empty, i.e. has no front subterm then
10    | fail inconsistent trail input literals
11  | else // refine the current variable
12    | let  $x$  be the current variable corresponding to the front subterms in  $\mathcal{Q}$ 
13    | partition  $\mathcal{Q}$  into disjoint subsets  $\mathcal{Q}_f$  according to their front symbols, i.e., the
      | top symbols of their front subterms, so that  $\mathcal{Q} = \bigsqcup_{f/n \in \Omega} \mathcal{Q}_f$ 
14    | foreach  $f/n \in \Omega$  do
15      | create a child node  $u_f$  of  $u$  with an edge labeled by  $\{x \mapsto f(y_1, \dots, y_n)\}$  for
        | fresh variables  $y_1, \dots, y_n$ 
16      | initialise  $\text{status}(u_f) \leftarrow \text{unfinished}$ 
17      | unfold the front subterm for each subterm view in  $\mathcal{Q}_f$ 
18      | recurse with BuildModel( $\mathcal{T}, u_f, \mathcal{Q}_f$ )
19    | end
20  | end
21 end

```

Every node is passed to BuildModel at most once. During a call, the procedure only changes the labels of its current node and introduces children below that node. Thus, the recursive execution has a well-defined tree and view state at every call.

In Algorithm 1, note that line 2 implements the Clean rule as given in the paper: It assigns a positive polarity by default. In the actual implementation, we however store such atoms with a special, undefined status. This allows us to choose the polarity of such an atom retroactively at any later point, while still retaining the guarantee that $\mathcal{T}$ is a model for Γ . For example, during clause evaluation (Section 4), this allows us to choose a polarity depending on the overall clause structure.

Correctness. At every recursive call, the views represent precisely the trail literals matching the current leaf. The three cases of the algorithm therefore implement *Clean*, *Solve*, and *Refine*, respectively. In the refinement case, all views agree on every previously advanced variable, and

the current variable is the first one on which the matching literals differ. Partitioning by top symbol produces exactly the refined branches; unfolding preserves the view invariant in every nonempty branch. Consequently, the procedure computes the result of the abstract calculus [5] when *Refine* always selects the first variable in a fixed breadth-first order. By induction over the recursive calls, the finished leaves form the resulting complete linear interpretation.

Theorem 1 (Efficient Linear Model Building). *Let Γ be a consistent set of ground literals of a predicate P/m and let $\mathcal{Q}$ be their subterm views. Then Algorithm 1 constructs a refinement tree representing the complete linear model of Γ generated by the calculus of Bromberger et al. [5]. Its runtime is $O(\text{size}(\mathcal{Q})|\Omega|)$ for nonempty $\mathcal{Q}$ and constant for empty $\mathcal{Q}$.*

Proof sketch. Correctness is the correspondence established above. For runtime, every subterm view advances through each entry at most once. Unfolding removes one entry and appends its immediate arguments. Hence, the number of nonempty recursive calls that perform a refinement is bounded by $O(\text{size}(\mathcal{Q}))$. Each such call processes its set of subterm views and explicitly creates one child for every symbol in Ω . All other operations over nonempty sets of subterm views take $O(\text{size}(\mathcal{Q}))$ time in total. The resulting runtime is therefore $O(\text{size}(\mathcal{Q})|\Omega|)$. If $\mathcal{Q}$ is empty, the procedure applies *Clean* in constant time. $\square$

For a trail containing several predicates, this construction is applied independently to the literals of each predicate. The following wrapper merely collects the resulting refinement trees. We call a collection containing one refinement tree for each predicate a *refinement forest*. In the following, we identify a linear model with its refinement-forest representation and use $\mathcal{M}$ for both.

Algorithm 2: BuildModel(Γ)

Input: A consistent set Γ of ground literals

Output: A refinement forest $\mathcal{M}$ representing a linear model of Γ

```

1 initialize an empty refinement forest  $\mathcal{M}$ 
2 for each predicate  $P/m \in \Pi$  do
3   initialize a refinement tree  $\mathcal{T}$  with a root  $r$  for  $P/m$ , where
     atom( $r$ ) =  $P(x_1, \dots, x_m)$ , and set status( $r$ )  $\leftarrow$  unfinished
4   let  $L_1, \dots, L_k$  denote the literals of  $P/m$  in  $\Gamma$ 
5   for each  $L_i$ , initialize its subterm view  $q_i$  as described above, and let
      $\mathcal{Q} = \{q_1, \dots, q_k\}$ 
6   call BuildModel( $\mathcal{T}, r, \mathcal{Q}$ )
7   add  $\mathcal{T}$  to  $\mathcal{M}$ 
8 end
9 return  $\mathcal{M}$ 

```

The wrapper adds constant work for each predicate. Each trail literal belongs to exactly one refinement tree. Overall, summing up the independent runtime of all procedures gives a runtime bound of $O(\text{size}(\Gamma)|\Omega| + |\Pi|)$.

4 Clause Evaluation

The question we address here is the following: given a linear model $\mathcal{M}$ and a clause C , is C false in $\mathcal{M}$? In other words, is there a grounding substitution σ such that for every literal $L\sigma \in C\sigma$ there is a complementary literal $K \in \mathcal{M}$ such that $\text{atom}(K\tau) = \text{atom}(L\sigma)$ for some grounding substitution τ ? If the literals in C do not share variables, this question can be answered in

polynomial time because all literals of C can be investigated independently. If, however, the literals in C share variables, clause evaluation becomes NP-complete, as shown below.

As a simple example, consider the clause $C = R(x, g(y)) \vee R(y, g(z))$ and the ground literals $\{\neg R(a, g(b)), \neg R(a, g(a))\} \subseteq \mathcal{M}$, with all other ground atoms positive in $\mathcal{M}$. If we match $R(x, g(y))$ with $R(a, g(b))$ under the substitution $\{x \mapsto a, y \mapsto b\}$, then the second literal $R(y, g(z))\{x \mapsto a, y \mapsto b\}$ is positive in $\mathcal{M}$, and C is not detected as false. However, if we start by matching $R(x, g(y))$ with $R(a, g(a))$, then the clause can be falsified by additionally choosing $z \mapsto b$. The situation is even more complex because literals in $\mathcal{M}$ may also contain variables.

Lemma 2 (Clause Evaluation is NP-Complete). *Let $\mathcal{M}$ be a linear model and C a clause. Then checking whether C is false in $\mathcal{M}$ is NP-complete.*

Proof. Let $C = L_1 \vee \dots \vee L_n$. For membership in NP, nondeterministically select n variable-disjoint copies of literals $K_1, \dots, K_n$ from $\mathcal{M}$ that have opposite signs, i.e., if L_i is positive, then K_i is negative, and vice versa. Now solve the unification problem $\{\text{atom}(L_1) = \text{atom}(K_1), \dots, \text{atom}(L_n) = \text{atom}(K_n)\}$. C is false iff the unification problem has a solution.

For hardness, we reduce from 3-SAT. Represent a 3-SAT clause $P_4 \vee \neg P_2 \vee P_7$ by a literal $\neg R(x_4, g(x_2), x_7)$, i.e., the clause is encoded by a 3-place relation R , positive propositional variables P_i by first-order variables x_i and propositional negation by a monadic function g . We assume two further constants: a for “true” and b for “false”. Furthermore, we need a monadic predicate S for excluding all terms that do not represent constant truth values. Then a 3-SAT clause set N is represented by one single clause $\neg S(x_1) \vee \dots \vee \neg S(x_n) \vee L_1 \vee \dots \vee L_m$, where the clause set N contains exactly the propositional variables $P_1, \dots, P_n$ and the literals L_j are translations of the clause C_j from N via the relation R . Finally, we need to construct the linear model $\mathcal{M}$. For the predicate S , it contains the literals $S(a)$, $S(b)$, and $\neg S(g(x))$. For the predicate R , it contains all literals with argument combinations of a , b , $g(a)$, and $g(b)$, where the respective 3-SAT clause is true or false, respectively. For example, $R(a, b, b)$, $R(b, g(a), a)$, $\dots$ represent clauses that are true, and $\neg R(b, b, b)$, $\neg R(b, g(a), b)$, $\dots$ represent false 3-SAT clauses. In order to obtain a complete linear model $\mathcal{M}$, for the remaining R atoms covering nested occurrences of g , we add the positive atoms $R(g(g(x)), t_1, t_2)$, $R(t_1, g(g(x)), t_2)$, and $R(t_1, t_2, g(g(x)))$ for $t_1, t_2 \in \{a, b, g(a), g(b), g(g(y)), g(g(z))\}$, omitting duplicates modulo variable renaming and non-linear atoms. Note that these atoms do not play a role in the evaluation of the clause $\neg S(x_1) \vee \dots \vee \neg S(x_n) \vee L_1 \vee \dots \vee L_m$ because of the S guards. Now N is satisfiable iff the first-order clause obtained from N is false in $\mathcal{M}$. $\square$

The implementation relies on the refinement tree introduced in Section 3. Given a linear model $\mathcal{M}$ represented by one refinement tree for each predicate and a clause C that is checked for falsification, we traverse the literals of C in any order. At any point in time, the procedure keeps a substitution σ over the variables of C , where σ is initially empty. For every literal $L \in C$, we pick the refinement tree according to the predicate symbol in L . Then the leaves of the tree with appropriate polarity that are unifiable with $L\sigma$ generate extensions of σ that can then be checked recursively for the remaining literals. The top algorithm looks as follows.

Algorithm 3: EvaluateClause($\mathcal{M}, C, \sigma$)

Input: A refinement forest $\mathcal{M}$ containing a tree for every predicate in the clause C and a substitution σ

Result: Returns true iff some grounding instance of $C\sigma$ is false in $\mathcal{M}$

```

1 if  $C \neq \perp$  then
2   let  $C = L \vee C'$ 
3   let  $\mathcal{T}$  be the refinement tree in  $\mathcal{M}$  corresponding to the predicate of  $L$ 
4   substs = EvaluateLiteral( $\mathcal{T}, L, \sigma$ )
5   foreach  $\sigma' \in \text{substs}$  do
6     if EvaluateClause( $\mathcal{M}, C', \sigma'$ ) then
7       return true
8     end
9   end
10  return false
11 return true

```

At the start, the EvaluateClause algorithm is called with an empty σ . The EvaluateLiteral subalgorithm produces a set of substitutions that are extensions of σ and can then be checked for the remaining literals. It works as follows.

Algorithm 4: EvaluateLiteral($\mathcal{T}, L, \sigma$)

Input: A refinement tree $\mathcal{T}$ for the predicate of L and a substitution σ

Result: A set of substitutions falsifying L with respect to $\mathcal{T}$

```

1 substs =  $\emptyset$ 
2 foreach leaf node  $u$  of  $\mathcal{T}$  do
3   if  $u$  has opposite polarity to  $L$  and  $\tau = \text{mgu}(\text{atom}(u), \text{atom}(L\sigma))$  exists then
4      $\sigma' = \sigma\{x \mapsto x\tau \mid x \in \text{vars}(\text{atom}(L)\sigma), x\tau \neq x\}$ 
5     substs =  $\{\sigma'\} \cup \text{substs}$ 
6 end
7 return substs

```

The substitutions returned by EvaluateLiteral represent all falsifying ground instances: every grounding instance of a returned substitution falsifies L , and every grounding extension of σ that falsifies L is an instance of a returned substitution.

Correctness of Algorithm 3 follows from the fact that the leaves of a refinement tree partition all ground atoms of its predicate. Hence, unifying $L\sigma$ with all leaves of opposite polarity yields precisely the substitutions whose grounding instances falsify L . By induction on the number of literals, the algorithm then returns true exactly if one common grounding instance falsifies every literal of the clause.

The actual implementation is a refinement of the above two abstract algorithms. The EvaluateLiteral algorithm delivers candidate substitutions in a lazy, iterative way, saving space for the set construction and substitutions that need not be traversed in case of success.

In addition, the EvaluateLiteral algorithm explores the structure of the refinement tree as a filter to rule out leaf nodes that cannot contribute to extending σ . The refinement tree is traversed following the structure of $\text{atom}(L)\sigma$. The variable refined at an internal node is identified by the domain of its outgoing edge substitutions. The subterm view implicitly keeps track of the corresponding variable of the current refinement-tree atom and provides the respective subterm of $\text{atom}(L)\sigma$ as its front term. The implementation performs a case analysis with respect to the current node u of the refinement tree and the corresponding current front

term $t\sigma$ of $\text{atom}(L)\sigma$:

- if u is not a leaf node
 - if $t\sigma$ is a variable: collect all leaf nodes representing literals with appropriate polarity; for each node whose atom unifies with $\text{atom}(L)\sigma$, yielding most general unifier τ , build $\sigma' = \sigma\{x \mapsto x\tau \mid x \in \text{vars}(\text{atom}(L)\sigma), x\tau \neq x\}$
 - if $t\sigma$ is not a variable: consider the child node v of u introducing the top function symbol of $t\sigma$; move to the respective subterm front of $t\sigma$
- if u is a leaf node:
 - if the literal of u has appropriate polarity and the atom of u unifies with $\text{atom}(L)\sigma$, yielding most general unifier τ , build $\sigma' = \sigma\{x \mapsto x\tau \mid x \in \text{vars}(\text{atom}(L)\sigma), x\tau \neq x\}$

An important invariant of the computation of the σ' is that all variables in $\text{atom}(u)$ are fresh. Refinement trees need not store any variables at all because of the linearity assumption. Thus, it is sufficient to store the maximal variable in σ with respect to some ordering and always use larger variables when $\text{atom}(u)$ is generated. More concretely, variables are represented by natural numbers, making these operations highly efficient.

5 Integrating into SPASS(SCL)

The SCL(FOL) calculus builds a ground literal partial model Γ , the trail [7]. At any point, the number of ground literals that may enter Γ is finitely limited. The calculus searches for a contradiction with respect to Γ , i.e., a clause C with grounding σ such that $C\sigma$ is false in Γ . This means for every literal $L\sigma \in C\sigma$ the complementary literal is contained in Γ . A clause is propagating a literal $L\sigma$ if neither $L\sigma$ nor its complement is contained in Γ and $C\sigma \setminus \{L\sigma\}$ is false in Γ . The detection of propagating and false literals is supported by a lifting from the CDCL 2-watched literal approach [22, 16] to first-order logic [4].

The construction of Γ then either ends with a conflict (the detection of a false clause) or a stuck state. For the latter there is no conflict with respect to Γ and the given clause set. Now we use the partial ground model Γ to build an overall linear model $\mathcal{M}$ via the model building procedure of Section 3. Then we check whether any clause from the input clause set is false with respect to $\mathcal{M}$, using the procedure of Section 4. If no false clause is detected, $\mathcal{M}$ is an overall model and satisfiability of the clause set is detected. Otherwise there is a clause C and grounding σ such that $C\sigma$ is false in $\mathcal{M}$. Obviously, there must be literals in $C\sigma$ that are undefined in the trail of the stuck state Γ . Then we extend the finite set of considered ground literals by exactly these ground literals and restart the search for a stuck state or a contradiction.

The next section, Section 6, reports on experiments using this set-up.

6 Evaluation

We evaluate the implementation both in isolation and as part of the overall SPASS(SCL) solver. First, we experimentally support the linear runtime bound from Theorem 1. Afterwards, we analyze the runtime and result impact of linear model building in the context of a full SPASS(SCL) run.

Problem	Building	Evaluation	Solver total	SZS status	Combined
SYN316+1	6.86%	1.06%	30.511s	Timeout	7.92%
SWV418-1.900	0.89%	7.10%	30.186s	Timeout	7.99%
SYO682-1	0.00%	8.49%	4.052s	Unsatisfiable	8.50%
SYN541-1	1.94%	6.91%	2.746s	Satisfiable	8.85%
SYN884-1	8.52%	0.45%	7.193s	Unsatisfiable	8.97%
NLP035-1	0.69%	11.51%	30.867s	Timeout	12.21%
SYN868-1	14.67%	1.38%	1.813s	Satisfiable	16.05%
SYO581+1	0.42%	19.89%	4.412s	Satisfiable	20.31%
NLP103-1	0.18%	26.61%	30.522s	Timeout	26.79%
NLP102-1	0.12%	27.08%	30.204s	Timeout	27.20%

Table 1: The ten nontrivial problem instances with the largest fraction of solver runtime spent on model building and clause evaluation.

6.1 Experimental Setup

We evaluate the implementation in the SPASS(SCL) solver. The solver was compiled with GCC 12.2.0 using the C99 standard and the optimization flags `-O3 -flto -fwhole-program`. The experiments were run single-threaded on a PowerEdge R7625 server equipped with two AMD EPYC 9754 processors and 2.21 TiB of DDR5 memory, running Debian GNU/Linux 12. Our problem set consists of all equality-free FOF and CNF problems from version 9.2.1 of the TPTP library [23]. Each problem was given a time limit of thirty seconds.

6.2 Linear Scaling of Model Building

To assess the practical scaling of model construction, we record the runtime of each invocation of `BuildModel` together with the size of the current trail, measured as the sum of the sizes of its literals. During a solver run, these invocations are performed on progressively larger trails. The resulting measurements, shown in Figure 2, show a clear linear correlation between trail size and model building time for the same problem, supporting the linear-time bound of Theorem 1. Measurements from different problems are not comparable: the bound in Theorem 1 depends on $|\Omega|$, which varies with the problem signature, as does the term structure.

6.3 Runtime Overhead

To assess the runtime overhead of the model procedures, we focus on nontrivial runs with a total runtime of at least one second and at least 10 milliseconds spent on model building and clause evaluation. On average, model building accounts for 0.39% of the solver runtime and clause evaluation for 0.42%, giving a combined overhead of 0.81%. Table 1 shows the ten instances with the largest fraction of time spent on model building and clause evaluation. Overall, model building and clause evaluation together almost always account for less than 10% of the solver runtime. Even for the largest outliers, their combined share remains below 30%.

6.4 Impact on SPASS(SCL)

Table 2 shows how the use of the model procedures relates to the overall outcome of a solver run. Linear model building is currently invoked in more than half of the overall SPASS(SCL) runs. Whenever clause evaluation finds a false instance, the considered ground literals are extended as described in Section 5. For 251 problems, those new considered literals ultimately

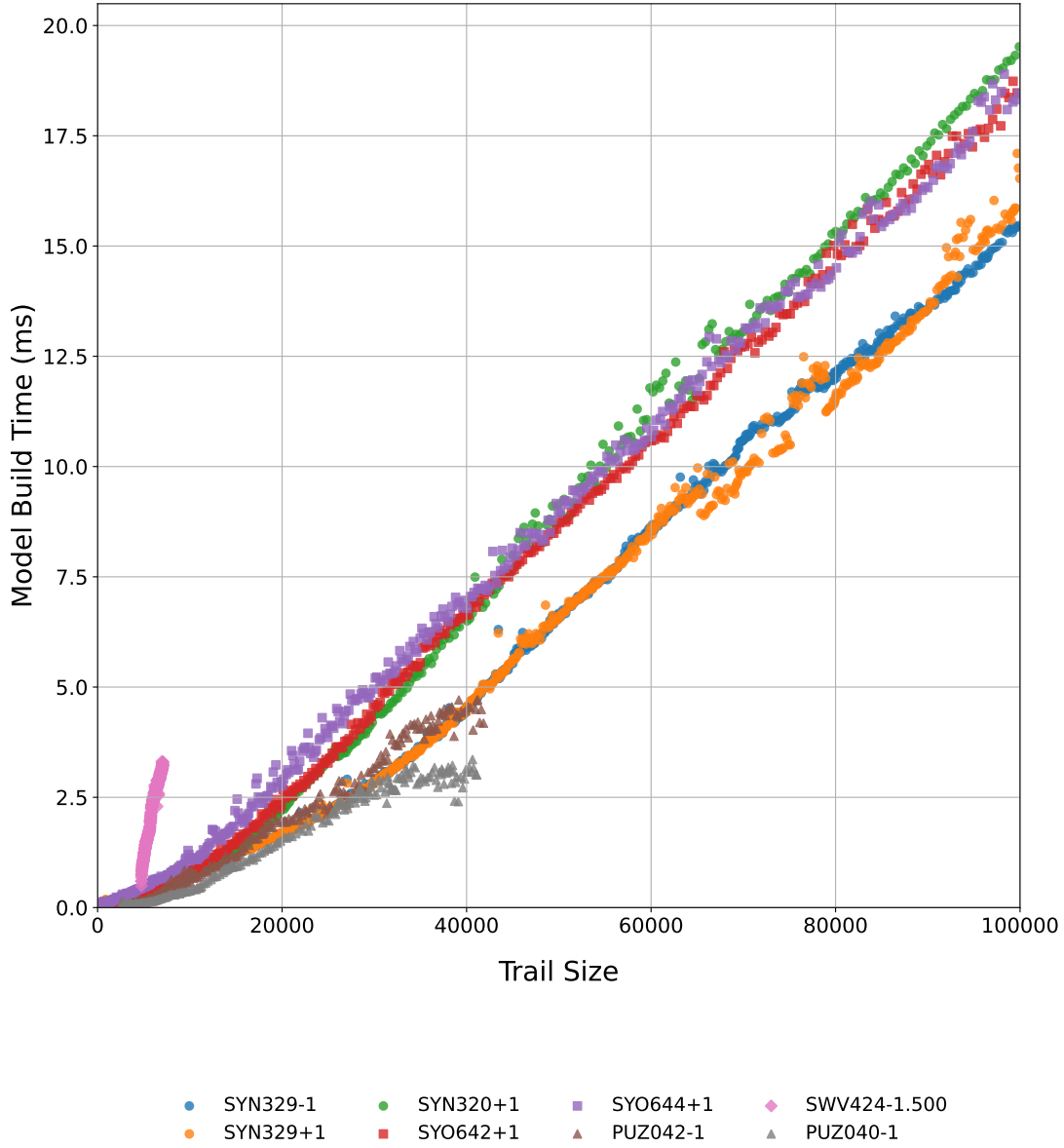


Figure 2: Model building time as a function of trail size.

lead to an unsatisfiability proof within the time limit. Linear model building not only provides counterexamples that guide the subsequent proof search, but can also establish satisfiability directly.

Besides simple preprocessing, linear model building is currently the only approach to establish satisfiability of a clause set in SPASS(SCL). Out of the 1136 satisfiable instances in the considered problem set, 327 are solved by our linear model building. Thus, linear model building and the subsequent clause evaluation already detect about 29% of all satisfiable instances.

Problem set	Count	Unsat.	Sat.	Timeout
all	4232	1550	327	2355
└ model procedures not called	1963	1299	0	664
└ model procedures called	2269	251	327	1691

Table 2: Solver results on the considered TPTP problems, partitioned by whether the model procedures were invoked during the run.

7 Conclusion

We presented concrete data structures and algorithms for building a complete linear model from a finite trail of ground literals, implementing previous theoretical work [5]. Moreover, we gave an algorithm for evaluating clauses in the resulting interpretation. To this end, we introduce refinement trees which represent the structure of the abstract model generation calculus efficiently. Subterm views provide direct access to the term positions needed during model building and clause evaluation. This results in a model building algorithm with runtime linear in the size of the trail for a fixed signature.

Although clause evaluation is worst-case NP-complete, in our experiments we don’t see an exponential run-time behavior. The time for clause evaluation can actually be neglected compared to the time SPASS(SCL) needs for running the rest of the calculus. This is not too surprising because also for hyper-resolution [19, 13, 20, 24] the worst-case exponential behavior does typically not show up. An efficient implementation is nevertheless mandatory. In the case of clause evaluation, lazy traversal and backtracking avoid the materialization of intermediate substitutions and allow the traversal state to be restored in place.

Our experimental evaluation confirms that the algorithms can be efficiently implemented. Model building scales linearly with growing trails in practice. Within the SPASS(SCL) solver, on nontrivial runs, model building and clause evaluation together account for less than one percent of the overall runtime on average. This time is well-spent: If the generated interpretation is a model of the input clauses, satisfiability is established directly. Otherwise, clause evaluation produces a concrete counterexample that can be used to extend the set of considered ground literals. In our evaluation, 327 satisfiable problems are detected through the generated linear models.

The restriction to linear Herbrand-style models remains a limitation. This can be seen on the SYN316* problems in our evaluation. Up to renaming, the satisfiable CNF problem SYN316-1 consists of the clauses

$$\neg P(x) \vee \neg P(s(x)) \quad \text{and} \quad P(x) \vee P(s(x)).$$

They require the truth value of P to alternate along every s -chain. The clause set has a model with two elements where s exchanges the elements. In a ground Herbrand model, however, the truth value alternates at unbounded term depth, so no finite set of linear literals can represent such a model. Our procedure can still generate counterexamples and guide the growth of the trail, but it cannot establish satisfiability for this problem.

Representing infinite Herbrand models via atoms (ARMs) is well-known [8]. Effectively and efficiently generating such models out of stuck SCL states is our contribution. There are a number of approaches towards more expressive Herbrand model representations known [2, 18, 1] that might be extendable to our context. There are also a number of approaches towards finite Herbrand model representations [15, 9, 12, 3], however they cannot be used to guide SCL towards additional ground literals.

Acknowledgements We thank our anonymous reviewers for their constructive comments.

References

- [1] Gábor Alagi and Christoph Weidenbach. NRCL - A model building approach to the bernays-schönfinkel fragment. In Carsten Lutz and Silvio Ranise, editors, *Frontiers of Combining Systems - 10th International Symposium, FroCoS 2015, Wrocław, Poland, September 21-24, 2015. Proceedings*, volume 9322 of *Lecture Notes in Computer Science*, pages 69–84. Springer, 2015.
- [2] Hicham Bensaid, Ricardo Caferra, and Nicolas Peltier. Dei: A theorem prover for terms with integer exponents. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 146–150. Springer, 2009.
- [3] Stefan Borgwardt and Barbara Morawska. Finding finite herbrand models. In Nikolaj S. Bjørner and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings*, volume 7180 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2012. Extended Version LTCS-Report 19-07.
- [4] Yasmine Briefs, Martin Bromberger, Tobias Gehl, Lorenz Leutgeb, Simon Schwarz, and Christoph Weidenbach. A two-watched literal scheme for first-order logic. In Armin Biere, Carsten Lutz, and Sara Negri, editors, *Automated Reasoning - 13th International Joint Conference, IJCAR 2026, Lisbon, Portugal, July 26-29, 2026, Proceedings, Part II*, volume 16689 of *Lecture Notes in Computer Science*, pages 20–36. Springer, 2026.
- [5] Martin Bromberger, Florent Krasnopol, Sibylle Möhle, and Christoph Weidenbach. First-order automatic literal model generation. In *Automated Reasoning – 12th International Joint Conference, IJCAR 2024*, volume 14739 of *Lecture Notes in Computer Science*, pages 133–153. Springer, 2024.
- [6] Martin Bromberger, Simon Schwarz, and Christoph Weidenbach. Exploring partial models with SCL. In Ruzica Piskac and Andrei Voronkov, editors, *Proceedings of the 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 94 of *EPiC Series in Computing*, pages 48–72. EasyChair, 2023.
- [7] Martin Bromberger, Simon Schwarz, and Christoph Weidenbach. SCL(FOL) revisited, 2023.
- [8] Ricardo Caferra, Alexander Leitsch, and Nicholas Peltier. *Automated Model Building*, volume 31 of *Applied Logic Series*. Kluwer, 2004.
- [9] Koen Claessen and Niklas Sörensson. New techniques that improve mace-style finite model finding. In *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*, 2003.
- [10] Alberto Fiori and Christoph Weidenbach. SCL clause learning from simple models. In *Automated Deduction – CADE-27*, volume 11716 of *Lecture Notes in Computer Science*. Springer, 2019.
- [11] Peter Graf. *Term Indexing*, volume 1053 of *LNAI*. Springer, 1996.
- [12] Mikolas Janota and Martin Suda. Towards smarter mace-style model finders. In Gilles Barthe, Geoff Sutcliffe, and Margus Veanes, editors, *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*, volume 57 of *EPiC Series in Computing*, pages 454–470. EasyChair, 2018.
- [13] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 2013.
- [14] William McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *J. Autom. Reason.*, 9(2):147–167, 1992.

- [15] William McCune. Mace4 reference manual and guide. *CoRR*, cs.SC/0310055, 2003.
- [16] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001.
- [17] Robert Nieuwenhuis, Thomas Hillenbrand, Alexandre Riazanov, and Andrei Voronkov. On the evaluation of indexing techniques for theorem proving. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning, First International Joint Conference, IJCAR 2001, Siena, Italy, June 18-23, 2001, Proceedings*, volume 2083 of *LNCS*, pages 257–271. Springer, 2001.
- [18] Ruzica Piskac, Leonardo Mendonça de Moura, and Nikolaj Bjørner. Deciding effectively propositional logic using DPLL and substitution sets. *Journal of Automated Reasoning*, 44(4):401–424, 2010.
- [19] John Alan Robinson. Automatic deduction with hyper-resolution. *International Journal of Computer Mathematics*, 1:227–234, 1965.
- [20] Stephan Schulz, Simon Cruanes, and Petar Vukmirović. Faster, higher, stronger: E 2.3. In Pascal Fontaine, editor, *Proc. of the 27th CADE, Natal, Brasil*, number 11716 in *LNAI*, pages 495–507. Springer, 2019.
- [21] R. Sekar, I.V. Ramakrishnan, and Andrei Voronkov. Term indexing. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 26, pages 1855–1900. Elsevier, 2001.
- [22] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In Rob A. Rutenbar and Ralph H. J. M. Otten, editors, *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, November 10-14, 1996*, pages 220–227. IEEE Computer Society / ACM, 1996.
- [23] Geoff Sutcliffe. The TPTP problem library and associated infrastructure. from CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.
- [24] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischniewski. SPASS version 3.5. In *Automated Deduction – CADE-22*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer, 2009.