On Satisfiability for Quantified Formulas in Instantiation-Based Procedures

Nicolas Voirol and Viktor Kuncak*

École Polytechnique Fédérale de Lausanne (EPFL), Switzerland,
`{firstname.lastname}@epfl.ch`

Abstract. Procedures for first-order logic with equality are used in many modern theorem provers and solvers, yet procedure termination in case of interesting sub-classes of satisfiable formulas remains a challenging problem. We present an instantiation-based semi-decision procedure defined on a fragment of many-sorted first-order logic that succeeds on certain satisfiable formulas even if they contain, for example, associativity axioms. The models for which our procedure terminates have finite ranges of function symbols. We expect that our procedure can be integrated into other instantiating procedures such as E-matching with little performance impact. Our procedure is also compatible with specialized verification techniques that enable efficient reasoning about pure higher-order recursive functions. We integrated our procedure into the Leon verification framework. The implementation is publicly available and has been evaluated on non-trivial benchmarks featuring higher-order programs with quantified contracts.

1 Introduction

Many software verification techniques rely on sound support of quantified propositions. However, handling quantifiers in first-order logic remains a significant challenge even for state-of-the-art theorem provers and solvers. Instantiation-based approaches have been successfully integrated into many Satisfiability Modulo Theories (SMT) solvers through techniques such as E-matching [10, 11], yet completeness results are difficult to achieve. Harder still is the question of satisfiability which arises in tools that support counter-example finding in addition to proof construction. Although completeness for models is indeed impossible, recent work has shown that certain interesting fragments of FOL with theories can be efficiently decidable without compromising on performance or effectiveness [3, 12].

Contributions. Our contributions are the following.

(1) We propose *an instantiation-based technique* for a syntactic fragment of pure first-order logic that enables model finding during proof search. Our approach

*This work is supported in part by the European Research Council (ERC) Project *Implicit Programming* and Swiss National Science Foundation Grant "Constraint Solving Infrastructure for Program Analysis".

explores the space of instantiations and associates to each unexplored path in the instantiation graph a condition under which it is irrelevant. At any given time, if a satisfying model exists such that no remaining path is relevant, the procedure can terminate and report a total model for the input formula. Furthermore, relevance checks for paths can be used to direct proof construction and prune large sections of the search space. We discuss the required bookkeeping for our procedure and how existing procedures could integrate it for model finding or instantiation guiding. Our fragment accepts quantification over infinite domains and specification of interesting properties such as associativity or idempotence. We show refutation-completeness of our procedure as well as completeness for range-finite models (*i.e.*, models where all functions have finite ranges).

(2) We describe the *integration of our technique for quantifier handling into the Leon verification framework* for pure higher-order functional programs over unbounded data types. The instantiation procedure we implemented supports quantifiers ranging over arbitrary theories and proofs remain sound, however we do not claim any completeness results over this extended fragment. Although first-order logic is sufficiently expressive to encode many interesting verification problems, Leon restricts the formula encoding of considered problems to decidable logical fragments in order to guarantee different theoretical properties such as counter-example completeness [7, 24]. Recent work has shown that higher-order functions can also be encoded into decidable formulas, thus preserving Leon's counter-example completeness [25]. However, quantifier support significantly improves the expressiveness of contracts on first-class functions and we show that certain fragments can still guarantee counter-example completeness for models where functions have finite range.

1.1 Motivating examples

We are interested in proving verification conditions for useful programs with first-class functions. Such contracts concern function-typed arguments, and therefore naturally require some notion of quantification in many cases. Given that verifying quantifier-free functions over simple data types in Leon is complete for counter-examples [25], we are interested in techniques that can report sound models while searching for a refutation (proof). The domains of non-parametric data types supported in Leon are often infinite, making use of finite model finding [20] less immediate. Leon itself provides inductive proof construction, an area in which SMT automation is active and still ongoing work [19].

Example 1 (Associativity). Satisfiable quantified clause sets with infinite Herbrand universes can be difficult for many techniques. Consider the clauses

$$f(f(x,y),z) \simeq f(x,f(y,z)),\ \ f(f(1,2),3) \not\simeq f(1,f(2,2))$$

where x, y, z are universally quantified over $\mathbb{Z}$. This example combines quantified uninterpreted functions with integers, which, without further restrictions, is not even semi-decidable. However, in the absence of integer theory symbols (a property checkable syntactically), our technique extends to this fragment as well. E-matching will generate the two ground clauses

$$f(f(1,2),3) \simeq f(1,f(2,3)) \quad \text{and} \quad f(f(1,2),2) \simeq f(1,f(2,2))$$

and give up as the set of ground clauses remains satisfiable. Certain superposition-based techniques with redundancy elimination will saturate the clause set but have no easy means of extracting a model. Techniques that explore relevant subsets of the Herbrand universe such as [12] will enumerate a (satisfiable) superset of the infinite sequence starting with

$$\begin{gathered} f(f(1,2),f(2,3)) \simeq f(1,f(2,f(2,3))) \\ f(f(1,2),f(2,f(2,3))) \simeq f(1,f(2,f(2,f(2,3)))) \end{gathered}$$

and never terminate. Automatically constructing a model for the initial clause set short of enumeration is therefore not trivial. However, rather simple models do exist for this clause set, such as, for example:

$$f \rightarrow \{\, (1,1) \Rightarrow 1,\ (1,2) \Rightarrow 2,\ (1,3) \Rightarrow 3,\ (2,1) \Rightarrow 2,\ (2,2) \Rightarrow 1,\ (2,3) \Rightarrow 3,\ \text{else} \Rightarrow 0 \,\}.$$

The intuition behind our technique is that if a model exists such that the set of values of all ground terms in that model is finite, then there must exist a model where all functions have finite ranges. This observation enables us to ensure model existence when we can guarantee that the interpretation of the Herbrand universe is finite. In the above example, let us consider the relevant domain $D = \{1,2,3\}$ for each argument of f. If we can find a model M for the clause set such that $M(f)(i,j) \in D$ for $i,j \in D$, then all Herbrand term interpretations in M must fall into D as well and a model for the clause set therefore exists. Note that the model given above satisfies this intuitive requirement. Our technique applies this informal observation in a principled manner to obtain a sound procedure that is complete for models where functions have finite ranges.

Example 2 (Binary idempotence). Another example of difficult satisfiable clauses that fit into our fragment is

$$\begin{gathered} f(x,y) \simeq f(y,x), \quad f(x,y) \simeq f(x,f(x,y)), \\ f(f(1,2),2) \not\simeq f(1,f(2,f(1,3))) \end{gathered}$$

where x,y,z are again universally quantified. The quantified clauses specify commutativity and idempotence of the uninterpreted function symbol f, and again, satisfying models with small range for f exist.

Example 3 (Fold identity). Counter-examples with finite range functions arise during useful verification efforts, for instance when dealing with properties of higher-order fold operation over associative functions. The example in Figure 1 defines a generic list ADT with an associated content function that maps a given list into its element set and a higher-order fold function. We are interested to show that, given an associative and commutative operation f, the fold implementation ensures the order of the folded list is irrelevant. A natural question is whether the result of fold is the same even if we have merely the same *set* of elements (the same value of content). This conjecture does *not* hold, and our technique correctly produces the following counter-example:

```
    f ↦ { (A, A) ⇒ B, (B, A) ⇒ A, (A, B) ⇒ A, (B, B) ⇒ B, else ⇒ C }
list1 ↦ Cons(A, Cons(A, Nil()))
list2 ↦ Cons(A, Nil())
    a ↦ B
```

```
sealed abstract class List[A]
case class Cons[A](head: A, tail: List[A]) extends List[A]
case class Nil[A]() extends List[A]

def content[A](list: List[A]): Set[A] = list match {
  case Cons(head, tail) ⇒ Set(head) ++ content(tail)
  case Nil() ⇒ Set.empty[A] }

def fold[A](a: A, list: List[A], f: (A,A) ⇒ A): A = list match {
  case Cons(head, tail) ⇒ fold(f(a, head), tail, f)
  case Nil() ⇒ a }

def foldConjecture[A](list1: List[A], list2: List[A], a: A, f: (A,A) ⇒ A): Boolean = {
  require(content(list1) == content(list2) && forall { (x: A, y: A, z: A) ⇒
    f(f(x,y),z) == f(x,f(y,z)) && // associative
    f(x,y) == f(y,x) }) // commutative
  fold(a, list1, f) == fold(a, list2, f) }.holds
```

Fig. 1. Verification of fold identity properties. The **require** construct specifies a precondition on foldConjecture, and holds demands verification of the foldConjecture lemma. Leon handles Set through the theory of sets in CVC4 and through arrays in Z3.

2 Fragment and Procedure

We consider many-sorted first-order logic with equality. We consider a subset $\mathcal{U}$ of universally quantified formulas described as follows. Consider without loss of generality the formula F in conjunctive normal form, consisting of a conjunction of clauses $C_1 \wedge \cdots \wedge C_n$ where each C_k is a disjunction of literals $l_1 \vee \cdots \vee l_m$. All variables in F, usually denoted by x, y, z are taken as universally quantified. We use the letters a, b, c for constants and t, s, r for arbitrary terms. Terms consist of either constants, variables, or uninterpreted function applications whose arguments are terms. We will be interested in satisfiability of such formulas when sorts are interpreted over disjoint infinite sets.

Let $\mathcal{F} = \{f(\overline{s})_1, \ldots, f(\overline{s})_n\}$ the set of applications of uninterpreted function symbols in F (by convention, we associate to each $f(\overline{s})_i$ the function symbol f_i and arguments $\overline{s}_i$), and $\mathcal{V} = \{x_1, \ldots, x_m\}$ the set of universally quantified variables in F. Moreover, for each clause C_k, let $\mathcal{F}_k$ be the set of function applications in C_k and $\mathcal{V}_k = FV(C_k) \cap \mathcal{V}$ be the set of universally quantified variables appearing in C_k. We say $F \in \mathcal{U}$ iff the following hold:

1. no atom $x \simeq t[x]$ exists in F (modulo symmetry of $\simeq$) where $t[x]$ is a term depending on x and no atom $x \simeq y$ exists in F where $x, y \in \mathcal{V}$,
2. for each C_k, for each $x \in \mathcal{V}_k$ there must exist $f(\overline{s}) \in \mathcal{F}_k$ such that $x \in \overline{s}$.

Namely, constraint No. 1 ensures equality with quantified variables is limited to ground or independent terms, and constraint No. 2 states that all quantified variables in a clause must appear in some argument position.

2.1 Instantiation procedure

Our procedure performs quantifier instantiations guided by relevant uninterpreted function domains. Function applications taking quantified arguments (called matchers heareafter) are grouped into sets for each clause (matcher quorums) such that instantiations can be efficiently performed given a clause and set of ground applications. In order to ensure refutation-completeness, instantiation with a set of applications taking arguments in $\mathcal{V}$ is sometimes necessary. These variables are taken existentially in the resulting clauses and are considered ground when discussing instantiation. The procedure tracks an increasing set of ground applications that are used to generate ground clauses from quantified ones, and a set of *future* applications that will appear in the set of ground applications at some point. This set enables us to find certain satisfying models even when the set of instantiations is not finite.

We start by formalizing the above notion of matcher quorums. Consider $\mathcal{M}_k = \{f(\overline{x}) \in \mathcal{F}_k \mid \overline{x} \cap \mathcal{V}_k \neq \emptyset\}$ the set of all uninterpreted function symbol applications in C_k that have at least one universally quantified argument, and $\mathcal{V}_{k,f} = \{x \in \mathcal{V}_k \mid f(\overline{x}) \in \mathcal{M}_k, x \in \overline{x}\}$ be the set of quantified variables appearing in C_k as arguments to function symbol f. We inductively define the predicate $q_{k,f}$ as

$$q_{k,f}\,(M \subseteq \mathcal{M}_k) \iff \mathcal{V}_k \neq \emptyset \wedge \bigcup_{g(\overline{x}) \in M} \overline{x} \cap \mathcal{V} = \mathcal{V}_{k,f} \wedge \bigwedge_{S \subset M} \neg q_{k,f}(S).$$

Note that $q_{k,f}$ only holds for application sets that share a unique function symbol as it would otherwise hold for some subset. Furthermore, $q_{k,f}(\emptyset)$ never holds as $\mathcal{V}_k = \emptyset$ is disallowed and $\mathcal{V}_k \neq \emptyset$ implies $\{x_i \mid f(\overline{x}) \in M, x_i \in \overline{x}\}$ cannot be empty. Given $S_k = \{f \mid f(\overline{s}) \in \mathcal{M}_k\}$, the set of function symbols that take quantified arguments in C_k, and $\mathcal{Q}_{k,f} = \{M \subseteq M_k \mid q_{k,f}(M)\}$ the set of per-function symbol quorums, we define the set of quorums for C_k as

$$\mathcal{Q}_k = \left\{ \bigcup\nolimits_{f \in S_k} M_f \in \mathcal{Q}_{k,f} \mid \forall f(\overline{x}) \in M_f, g(\overline{y}) \in M_g . f \neq g \implies \overline{x} \cap \mathcal{V}_k \cap \overline{y} = \emptyset \right\}.$$

Note that each quorum uniquely defines the associated clause and can therefore provide efficient unification by building a mapping between the matchers in both clauses.

For a quorum $\{f(\overline{x})_1, \ldots, f(\overline{x})_n\}$, one generates instantiations given ground applications $f(\overline{s})_i$ for $1 \leq i \leq n$ by building a substitution θ that binds all variables $x \in \bigcup_{i=1}^{n} \overline{x}_i \cap \mathcal{V}$. Since we handle clause verification through an underlying solver, we also construct a condition c under which θ is a valid unifier. This condition encodes equality of the multiple images for quantified variables appearing in multiple argument positions in the quorum and ensures ground portions of $\overline{x}_i$ are equal to the corresponding ground terms in $\overline{s}_i$. The ground clauses we generate will therefore have the shape $c \implies \theta[\![C_k]\!]$. We let $\mathit{subst}(\{(f_1, \overline{x}_1, \overline{s}_1), \ldots, (f_n, \overline{x}_n, \overline{s}_n)\}) = (c, \theta)$

A non-trivial question is how to boostrap the instantiation process: what is the initial set of ground applications on which the quantified formulas should be instantiated? It is clear that all ground function applications in $\mathcal{F}$ must be considered here. However, we cannot ignore the ground portions of non-ground applications, as well as the intrinsic structure given by quantifier repetitions. We therefore consider all non-ground applications in $\mathcal{F}$ as well ($x \in \mathcal{V}$ will be existentially quantified in generated quantifier-free clauses). Finally, we must also consider literals of the shape $x \simeq t$ and $\neg(x \simeq t)$ where $x \in \mathcal{V}$ as t is clearly relevant to the domain of functions taking x as argument. Note that if $x \simeq t$ appears in C_k, it is actually terms $s \not\simeq t$ that are relevant to application $f(\overline{x})$ with $x \in \overline{x}$. We therefore ensure relevance of ground arguments by introducing fresh constants $a_{k,x}$ uniquely defined for each $x \in \mathcal{V}_k$ and ensuring $a_{x,k} \not\simeq t_i$ for each literal $x \simeq t_i$ in C_k. We can thus define the (finite) sets of initial applications $\mathcal{G}$ through the fixpoint relation

$$\begin{gathered}
\mathcal{F} \subseteq \mathcal{G} \\
\{f(x_1, \ldots, t, \ldots, x_n) \mid f(\overline{x}) \in \mathcal{G} \wedge x_i \in \mathcal{V}_k \wedge x_i \not\simeq t \in C_k\} \subseteq \mathcal{G} \\
\{f(x_1, \ldots, a_{k,x_i}, \ldots, x_n) \mid f(\overline{x}) \in \mathcal{G} \wedge x_i \in \mathcal{V}_k \wedge x_i \simeq t \in C_k\} \subseteq \mathcal{G}.
\end{gathered}$$

Instantiating non-ground applications will ensure that argument structure is preserved and matcher domains are never empty, however it does not explore relevant domains in a complete way. Indeed, consider the unsatisfiable clause set

$$f(x, a) \simeq a,\ f(b, y) \simeq b,\ \neg(a \simeq b)$$

where x, y are quantified and a, b are constants. The ground clauses generated through the substitution-based approach described above would consist in

$$a \simeq y \implies f(b, a) \simeq a \quad \text{and} \quad b \simeq x \implies f(b, a) \simeq b.$$

Clearly, extending the initial set of clauses with these two and taking a, b, x, y as existentially quantified is not equisatisfiable with the universally quantified case. We therefore provide a unification procedure for two function applications $f(\overline{x})$ and $f(\overline{y})$ where both applications are non-ground with ground portions and there exists $x_i \in \overline{x}$ such that $x_i \in \mathcal{V}$ and the corresponding y_i is ground. We build a substitution σ that binds every $x_j \in \mathcal{V}$ to corresponding y_j when it is ground. As in the case of θ above, we must also construct a condition c under which σ is valid to enable discharging to the solver. We let $\mathit{unify}(\overline{x}, \overline{y}) = (c, \sigma)$

and extend $unify$ to a total function by returning $(true, \{\})$ when the conditions on x_i and y_i existence are not met.

We define a sequence of triplets $(I_0, G_0, E_0), (I_1, G_1, E_1), \ldots$ where I_t is the set of ground matchers that have been instantiated at time t, G_t consists in the set of known future ground matchers that have not yet been considered and E_t is the set of clauses generated by the procedure. The full sequence is obtained through the inference rules defined in Figure 2, where

$$I_0 = \emptyset,\ G_0 = \{(true, f(\overline{s})) \mid f(\overline{s}) \in \mathcal{G}\}\text{, and}$$
$$E_0 = \{C_k \mid \mathcal{V}_k = \emptyset\} \cup \{a_{k,x} \not\simeq t \mid x \simeq t \in C_k\}\,.$$

$$\textsc{Inst}\ \frac{\begin{array}{c} q = \{f(\overline{x})_1, \ldots, f(\overline{x})_n\} \in \mathcal{Q}_k \qquad f(\overline{x}) \in \mathcal{M}_k - q \\ (b_i, f_i(\overline{s}_i)) \in I_t, 1 \leq i \leq n \qquad (c, \theta) = subst(\{(f_i, \overline{x}_i, \overline{s}_i) \mid 1 \leq i \leq n\}) \end{array}}{(\bigwedge_{i=1}^n b_i \wedge c,\ \theta[\![f(\overline{x})]\!]) \in G_t \qquad (\bigwedge_{i=1}^n b_i \wedge c \implies \theta[\![C_k]\!]) \in E_t}$$

$$\textsc{Unify}\ \frac{(b_s, f(\overline{s})), (b_r, f(\overline{r})) \in I_t \qquad (c, \sigma) = unify(\overline{s}, \overline{r})}{(\sigma[\![b_s]\!] \wedge c,\ \sigma[\![f(\overline{s})]\!]) \in G_t}$$

$$\textsc{Progress}\ \frac{(b, f(\overline{s})) = selection(G_t - I_t)}{I_{t+1} = I_t \cup \{(b, f(\overline{s}))\} \qquad G_t \subseteq G_{t+1}}$$

Fig. 2. Inferrence rules for the (I_t, G_t, E_t) sequence computation. The Inst rule ensures that all required instantiation clauses are generated and added to E_t and bookkeeping information appears in G_t. Unify makes sure that all relevant ground argument tuples are indeed considered, and the Progress rule guarantees that new instantiations take place as long as they exist. We call the choice of the new instantiation $(b, f(\overline{s})) \in G_t - I_t$ a *selection* such that $selection(G_t - I_t) = (b, f(\overline{s}))$.

Note that we leave the definition of the $selection(G_t - I_t)$ function open in the Progress rule, as long as it is fair. This allows heuristics such as E-matching where instantiations implied by the E-graph are given priority.

Lemma 1. *If F is satisfiable, then E_t is satisfiable for all t.*

Proof. The $\theta[\![C_k]\!]$ part of clauses generated by Inst consists in a substitution of quantified variables in C_k by ground terms, therefore $M \models C_k$ implies $M \models \theta[\![C_k]\!]$, and we have $M \models (c \implies \theta[\![C_k]\!])$ for arbitrary (ground) c.

Semi-decidability. We show that our procedure is refutation-complete by showing it implements the superposition calculus given by the following rules [1, 17].

$$\textsc{Resolution}\ \frac{l_L \vee l_1 \qquad l_2 \vee l_D}{\sigma[\![l_L \vee l_D]\!]}\ \sigma \text{ is } \mathsf{MGU}(l_1, \neg l_2)$$

$$\textsc{Equality Resolution}\ \frac{l_L \vee s \not\simeq s'}{\sigma[\![l_L]\!]}\ \sigma \text{ is } \mathsf{MGU}(s, s')$$

$$\textsc{Equality Factoring}\ \frac{l_L \vee s' \simeq t' \vee s \simeq t}{\sigma[\![l_L \vee t \not\simeq t' \vee s \simeq t']\!]}\ \sigma \text{ is } \mathsf{MGU}(s,s')$$

$$\textsc{Superposition Right}\ \frac{l_L \vee t \simeq t' \qquad l_D \vee s[u] \simeq s'}{\sigma[\![l_L \vee l_D \vee s[t'] \simeq s']\!]}\ \sigma \text{ is } \mathsf{MGU}(t,u),\ u \text{ non-variable}$$

$$\textsc{Superposition Left}\ \frac{l_L \vee t \simeq t' \qquad l_D \vee s[u] \not\simeq s'}{\sigma[\![l_L \vee l_D \vee s[t'] \not\simeq s']\!]}\ \sigma \text{ is } \mathsf{MGU}(t,u),\ u \text{ non-variable}$$

Lemma 2. *If applying superposition to F obtains the empty clause, then there exists a sequence of selections by the* Progress *rule*

$$(b_1, f(\overline{s})_1) \in G_0 - I_0, \ldots, (b_n, f(\overline{s})_n) \in G_{n-1} - I_{n-1}$$

such that $I_n = \bigcup_{i=1}^{n}\{(b_i, f(\overline{s})_i)\}$ and $\bigcup_{i=0}^{n} E_i$ is unsatisfiable.[1]

Theorem 1. *For any fair selection strategy by* Progress*, F and $\bigcup_i E_i$ are equisatisfiable.*

Proof. Lemma 1 gives us one direction. For the other, Lemma 2 tells us that if F is unsatisfiable, there is a finite set of selections by Progress such that the corresponding set of clauses is unsatisfiable. A fair selection strategy ensures this set of selections will eventually appear in I_t for some t, and application of the Compactness Theorem concludes our proof.

2.2 Extracting range-finite models

Interestingly, one can extend the above procedure to model generation. By convention, given an interpretation M, we write $M(v)$ for the value of symbol v in M, and $M[\![t]\!]$ for the interpretation of term t. Let us consider the clause set $\Sigma_t = \bigcup_{i=0}^{t} E_i$, model $M \models \Sigma_t$ and for each uninterpreted function symbol f in F, let $D_f = \{M[\![\overline{r}]\!] \mid (b_r, f_r(\overline{r})) \in I_t \wedge M \models b_r \wedge f = f_r\}$ be the relevant domain of f given by model M. If the set of relevant term interpretations is finite for the formula F, then a model with finite function ranges must exist.

Lemma 3. *If for $(b, f(\overline{s}))$ in G_t, either $M \not\models b$ or $M[\![\overline{s}]\!] \in D_f$, then $M \models E_{t+1}$.*[1]

Lemma 3 implies that chosing any $(b, f(\overline{s})) \in G_t - I_t$ to add to I_{t+1} will have no impact on satisfiability of E_{t+1}. Moreover, the same model that satisfied E_t will still hold for E_{t+1}, regardless of the chosen $(b, f(\overline{s}))$. Note that both conditions can be encoded into clauses that are given to the solver in conjunction with Σ_t.

Theorem 2. *If Lemma 3 holds for M and G_t, then there exists $M^F \models F$.*

Proof. Observe that if $M \not\models b$ or $M[\![\overline{s}]\!] \in D_f$ for all $(b, f(\overline{s}))$ in G_t, the property also holds for G_{t+1} (the arguments are similar to those exposed in Lemma 3 for E_{t+1}). We know $\Sigma_i \subseteq \Sigma_t$ for all $i \le t$, so by induction on t, we have $M \models \Sigma_j$ for $j \in \mathbb{N}$. The proof then follows by equisatisfiability of $\bigcup_i E_i$ and F.

[1] The complete proofs of Lemmas 2 and 3 can be found in Appendix A.

Theorem 2 implies soundness of our model finding technique, however we can further show completeness for models where all functions have finite ranges.

Theorem 3. *If M exists such that $|\{M(f)(\overline{s})\}|$ is finite for each uninterpreted function symbol f in formula F, then there exists t such that $M_t \models \Sigma_t$ and conditions of Lemma 3 hold for M_t.*

Proof. Let us consider $\Gamma = \bigcup_i G_i$ and assume that Γ has infinite cardinality (Lemma 3 trivially holds for finite Γ). Let $M \models \bigcup_i E_i$ with finite range for all function symbols. Given the Inst and Unify rules, there can only be a finite number of constants in Γ arguments. Furthermore, as all function symbols have finite domain in M, the set $\{M[\![\overline{s}]\!] \mid (b, f(\overline{s})) \in \Gamma\}$ must be finite as well. Since I_t is strictly increasing in t, the above set will eventually be coverable by $M_t \models \Sigma_t$, therefore M_t satisfying the conditions of Lemma 3 must eventually exist.

Relevance to other procedures. The procedure described in the previous section ensures that only a relevant subset of the Herbrand universe is included in G_t. However, tracking this subset can be expensive for certain instantiation procedures, so efficient representations or good approximations of G_t are essential. One possible approximation that can be easily computed based on trivial bookkeeping consists in a single expansion step of the Herband universe given the terms in scope. More formally, given the set of uninterpreted function symbols S_f and considered terms S_t, we define the single expansion step of the Herbrand universe in time i as

$$H_i = \{f(t_1, \ldots, t_n) \mid f \in S_f, t_1, \ldots, t_n \in S_t\}.$$

Clearly $\{f(\overline{s}) \mid (b, f(\overline{s})) \in G_t\} \subseteq H_t$. Therefore, if $M[\![\overline{s}]\!] \in D_f$ for all $f(\overline{s}) \in H_t$, then Theorem 2 holds for M as the second condition of Lemma 3 is satisfied for all $(b, f(\overline{s})) \in G_t$.

We generalize the above observation to a broader set of cases. Let us consider the sets S, T of condition $\times$ application tuples. We say $S \sqsubseteq T$ iff for all models M, the conditions of Lemma 3 on M holding for T implies they hold for S. Let us assume we have functions $\widehat{E}(I)$ and $\widehat{G}(I)$ such that for any $t \in \mathbb{N}$, we have $\widehat{E}(I_t) \iff \Sigma_t$, $G_t \sqsubseteq \widehat{G}(I_t)$, and the set of constants contained in terms of $\bigcup_t \widehat{G}(I_t)$ is finite. Note that selection of the $(b_t, f(\overline{s})_t) \in G_t - I_t$ by the Progress rule is only constrained by fairness, so any such selection criteria is valid. Furthermore, introducing $(b, f(\overline{s})) \notin G_t - I_t$ into I_t will have no impact on procedure soundness. Finally, consider the pair M_t, B_t such that $M_t \models \widehat{E}(I_t)$,

$$B_t = \left\{(b, f(\overline{s})) \in \widehat{G}(I_t) \mid (b, f(\overline{s})) \text{ does not satisfy conditions of Lemma 3}\right\},$$

and M_t minimizes $|B_t|$. Note that minimizing the cardinality of B_t is not altogether trivial, but techniques such as considering incrementally growing cardinalities or relying on unsat cores can provide precise computation techniques or efficient approximations for the M_t, B_t pair.

Corollary 1. *If for all t where M_t exists, there is a $j \geq t$ such that $B_t \subseteq I_j$, then the formulas $\bigcup_i \widehat{E}(I_i)$ and F are equisatisfiable. Furthermore, results of Theorems 2 and 3 also extend to $\widehat{E}(I_t)$ and $\widehat{G}(I_t)$.*

This result show that existing instantiation procedures can be extended with our technique when dealing with formulas in $\mathcal{U}$, even when they don't ensure refutation-completeness. For example, one can show that clauses generated by E-matching are equisatisfiable with Σ_t, and it is easy to see that the set $\widehat{G}(I_t) = \{(true, f(\overline{s})) \mid f(\overline{s}) \in H_t\}$ satisfies $G_t \sqsubseteq \widehat{G}(I_t)$. Another example is the Complete Instantiation procedure described in [12], which satisfies the constraints on both $\widehat{E}(I_t)$ and $\widehat{G}(I_t)$. Moreover, we show that one can guide quantifier instantiation based on failed model existence checks in order to ensure refutation-completeness of a procedure without explicitly instantiating the full Herbrand universe. Unlike MBQI presented in [12], the quantifier instantiation heuristic we propose considers all further instantiations jointly. This can improve the relevance of instantiations at the cost of performance.

Model construction. Given a model M that satisfies the two conditions proposed in Lemma 3, it is not entirely trivial to create a model satisfying F. We extend Σ_t with clauses that enforce desirable properties on the model, such as providing argument tuple instances *outside* of the relevant function domains and enforcing uniqueness of images for related tuples. We then given an *ite* construction based on conditions ϕ that ensures model soundness. For each function symbol f, we compute the set of relevant arguments $R_f = \{\overline{s} \mid (b, f(\overline{s})) \in G_t \wedge M \models b\}$ and projections $R_{f,i} = \{s_i \mid \overline{s} \in R_f\}$ for each argument position i. For each $R_{f,i}$, we consider the sets $X_{f,i} = R_{f,i} \cap \mathcal{V}$ and $S_{f,i} = R_{f,i} - X$. If M and $R_{f,i}$ for all f, i satisfy the rules in Figure 3, then sound models can be extracted.

$$\textsf{Equiv-1}\ \ X_{f,i} \subseteq \mathcal{X}_{f,i} \qquad \textsf{Ext}\ \ M(\mathcal{X}_{f,i}) \cap M(S_{f,i}) = \emptyset \qquad \textsf{Eq}\ \ |M(\mathcal{X}_{f,i})| = 1$$

$$\textsf{Equiv-2}\ \ \frac{X_{f,i} \cap X_{g,j} \neq \emptyset}{\mathcal{X}_{f,i} = \mathcal{X}_{g,j}} \qquad \textsf{Diff}\ \ \frac{\mathcal{X}_{f,i} \neq \mathcal{X}_{g,j}}{M(\mathcal{X}_{f,i}) \neq M(\mathcal{X}_{g,j})}$$

Fig. 3. Rules Equiv-1 and Equiv-2 ensure $\mathcal{X}_{f,i}$ is an equivalence class over all variables in $\mathcal{V}$ that appear in applications of f at position i. The Eq rule then ensures each equivalence class has a unique image, while Diff makes sure different classes have different images. Finally, the Ext rule ensures each class is indeed outside of the relevant function domain.

For each $\overline{s} \in R_f$, we define

$$\begin{aligned}
\textsf{EqGround}_{f,\overline{s}}(\overline{x}) &= \{\, x_i \simeq s_i \mid s_i \notin \mathcal{V} \,\},\\
\textsf{EqStruct}_{f,\overline{s}}(\overline{x}) &= \{\, x_i \simeq x_j \mid s_i, s_j \in \mathcal{V}, i \neq j \wedge M \models s_i \simeq s_j \,\},\\
\phi_{f,\overline{s}}(\overline{x}) &= \bigwedge \textsf{EqGround} \cup \textsf{EqStruct}.
\end{aligned}$$

The rules in Figure 3 ensure that for any two $\overline{s}_1, \overline{s}_2 \in R_f$ and terms $\overline{r}$ such that $M \models \phi_{f,\overline{s}_1}(\overline{r})$ iff $M \models \phi_{f,\overline{s}_2}(\overline{r})$, then $M[\![f(\overline{s}_1)]\!] \simeq M[\![f(\overline{s}_2)]\!]$. Consider the ordered sequence $\overline{s}_1, \ldots, \overline{s}_n$ of all $\overline{s}$ from R_f sorted by the inverse lexical ordering on the $(|\mathsf{EqGround}_{f,\overline{s}_i}(\overline{x})|, |\mathsf{EqStruct}_{f,\overline{s}_i}(\overline{x})|)$ pairs. We build model M^F for F by letting

$$M^F(f)(\overline{r}) = \begin{cases} M[\![\ \overline{s}_1\ ]\!] & \text{if } M \models \phi_{f,\overline{s}_1}(\overline{r}) \\ \quad\vdots & \quad\vdots \\ M[\![\ \overline{s}_{n-1}\ ]\!] & \text{if } M \models \phi_{f,\overline{s}_{n-1}}(\overline{r}) \\ M[\![\ \overline{s}_n\ ]\!] & \text{otherwise} \end{cases}$$

Note that the condition $\phi_{f,\overline{s}}(\overline{x})$ naturally entails an *ite* construction for function arguments $\overline{x}$. It is also plain to see that all constraints on M listed in Figure 3 can be encoded into satisfiability clauses and added to E_t to verify satisfiability of a quantified formula. Note that satisfying the constraints of Lemma 3 suffices to determine model existence. If they hold, then constraints on $R_{f,i}$ will eventually be satisfied for some $j \geq t$.

2.3 Supporting fragments with theories

We have already shown our model finding technique can be applied as it is to formulas that depart from many-sorted first-order logic with equality only (see examples in 1.1). Moreover, as the ground reasoning is handled by the SMT solver, ground portions of the input formula can involve arbitrary theory symbols soundly. One could, for instance, replace the (ground) second clause of Example 1 by $f(f(1,2),3) > f(1,f(2,2))$ and our technique would produce the same valid model. Thanks to the similarities in the instantiation procedures, our technique should extend to the fragment of *essentially uninterpreted formulas* defined in [12], along with *arithmetical literals* and *offsets*. This requires broadening our handling of $x \not\approx t$ literals in $\mathcal{G}$ to include $\neg(x \leq t)$ and $\neg(t \leq x)$ where t is a ground term. Furthermore, we introduce a preprocessing step such that for each literal $\neg(x_i \leq x_j)$ in C_k, we add the clause $\{x_j \rightarrow x_i\}C_k$ to F. Note that this is only required if x_i and x_j appear in different argument positions in C_k. Offsets are handled through a trivial extension to the Inst rule. Although our instantiations are slightly more precise, one can show that $\bigcup_i E_i$ and F^* are equisatisfiable. Indeed, given $M \models \bigcup_i E_i$ and projection functions $\pi_{f,j}$ and $\pi_{k,i}$ as defined in [12] with the additional constraint that $\pi_{f,1}(v_1), \ldots, \pi_{f,n}(v_n) \in D_f$ for all $\overline{v}$, one can see that $M^\pi \models F^*$. The model extraction procedure discussed in 2.2 can be adapted to the extended fragment by introducing fresh constants for relevant domain ranges (determined by $\neg(x \leq t)$, $\neg(t \leq x)$, and $\neg(x_i \leq x_j)$ literals). One must however note that the same limitations with respect to nonstandard models of arithmetic apply here, and satisfiability is not established modulo the class of intended structures.

Example 4 (Bounded quantification through guards). A notable difference between our procedure and that presented in [12] is the association of the boolean guards to each instantiation. Given the clause $\neg(a \leq x) \vee \neg(x \leq b) \vee f(x) < f(f(x))$, it is clear that the relevant term domain of f is infinite, and furthermore the second condition of Lemma 3 can never be satisfied for all members of G_t. However, by associating the guard $\neg(a \leq x) \vee \neg(x \leq b)$ to the $f(x)$ matcher, one can clearly satisfy the first condition of Lemma 3 for some interpretation of a and b. This illustrates how the instantiation guards not only ensure soundness of the procedure but can also increase the scope of the model finding technique.

3 Implementation

We implemented our technique inside the Leon verification system. The unfolding procedure underlying recursive function verification in Leon progressively considers a sequence of over- and under-approximations of the current verification condition. These approximations are encoded into quantifier-free formulas that are handled by an SMT solver. The formulas are instrumented in a way that offers control over the decision tree stemming from branching expressions in the program. Branches of the tree can be selectively dissallowed inside the formula. Finite counter-examples are generated by blocking all paths in the tree that depend on function calls that have not yet been unfolded. If, on the other hand, the verification condition is unsatisfiable regardless of the results of functions calls that still require unfolding, then no counter-example exists and we have a proof. For further discussions about unfolding in Leon, see [7, 24]. In order to handle first-class function applications, Leon dynamically dispatches the application to all relevant functions by creating a tree branch for each target function. By preserving decision tree soundness, this can be done independently from named function unfolding. See [25] for a complete formalization in the presence of first-class functions. Finally, quantifier instantiation takes place inside the unfolding loop as well, as shown in Figure 4. After each unfolding step, new ground function applications appear and clauses are generated for these. Note that a potentially infinite number of clauses can follow from finite ground applications, therefore only a heuristically determined subset of instantiations take place at each step. Completeness for range-finite counter-examples is ensured by the fairness of each independent instantiation procedure (named functions, first-class functions, and quantifiers).

We introduced a new `forall` construct that enables the specification of univerally quantified formulas in arbitrary non-nested positions inside Leon programs. Arbitrary positions are trivially handled through place-holder constants and lifting. Formula polarity is also of little consequence as the negation of a universally quantified formula consists in the negated formula taken existentially, which is exactly what our procedure produces when instantiating the non-ground applications from the formula. As Leon is not complete for proofs, we allow a broader fragment of quantified formulas than the one described in 2 since it suffices to produce valid instantiations for proofs to be sound. Fragment inclusion is therefore only determining when producing counter-examples. The implemented instantiation procedure is sligthly simpler than what we presented in 2.1 and the fragment on which we are complete for range-finite models is therefore somewhat restricted. We namely completely disallow equality with quantified variables (extending condition No. 1) and require arguments to function applications to

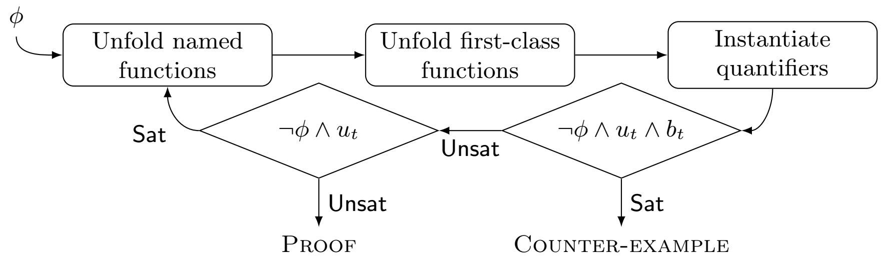

Fig. 4. Flow of the Leon verification procedure. The verification condition ϕ is verified by checking no counter-example exists. The procedure first checks $\neg\phi \wedge u_t \wedge b_t$ where u_t consists in the generated clauses and b_t disallows relevant tree branches. Note that b_t also contains the formula encoding of the conditions presented in Lemma 3. If no counter-example was found, then $\neg\phi \wedge u_t$ is considered in order to construct a proof. If this fails as well, we conclude that further instantiations are necessary.

either be fully ground or contain no ground portion. These restrictions remove the need for the Unify rule and simplify the computation of $\mathcal{G}$.

Formula instrumentation in Leon leads to clauses having the shape $b \implies c$ such that b encodes the current decision tree branch. Quantifier support can be naturally integrated by using the branch blocker b as condition for the ground blockers in c. This enables Leon to only consider relevant instantiations when constructing counter-examples. As Leon sits on top of the SMT solvers, the E-graph is not available to our procedure. The heuristics behind quantifier instantiation are therefore mostly syntax-based and rather conservative. Our initial evaluations show reasonable performance for proofs, but model finding becomes slow in the presence of over-instantiating as G_t blows up.

Experience. We have evaluated our implementation on various benchmarks involving higher-order functions as well as sets and arrays. Many of these benchmarks feature complex interactions between named functions, first-class functions and quantifiers.

Example 5 (Functional predicate map). As mentionned above, polarity of universally quantified formulas is soundly handled by our procedure, enabling a natural definition of existentials through the following named function:

```
def exists[A](p: A ⇒ Boolean): Boolean = !forall((a: A) ⇒ !p(a))
```

Based on this, one can define maps over predicates as shown in Figure 5 and verify their associativity. One should note here that our implementation can produce sound proofs even in those cases where quantifier nesting takes place. In order to ensure complete clause generation when first-class function bodies contain matchers, these must be axiomatized when applications taking quantified arguments exist in the formula..

```
def map[A,B](p: A ⇒ Boolean, f: A ⇒ B): B ⇒ Boolean =
  (b: B) ⇒ exists[A](a ⇒ p(a) && f(a) == b)

def equals[A](p: A ⇒ Boolean, that: A ⇒ Boolean): Boolean =
  forall[A](a ⇒ p(a) == that(a))

def functorConjecture[A,B,C](p: A ⇒ Boolean, f: A ⇒ B, g: B ⇒ C): Boolean = {
  equals(map(map(p, f), g), map(p, (a: A) ⇒ g(f(a)))) }.holds
```

Fig. 5. Verifying associativity of higher-order map definition through higher-order equality for functional predicates.

Example 6 (Binary Search Tree). Verifying that insertion into a binary search tree preserves the tree invariant proves a significant challenge in quantifier-free inductive verification. Quantifier support in Leon enables both an elegant specification of the property as well as solves the associated verification problem. Appendix B.1 shows the source code of this example. Leon verifies the property given the following inductive predicate isBST (trivial base case is omitted)

$$\begin{aligned}\mathsf{isBST}(\mathsf{Node}(L, v, R)) \iff \quad & \mathsf{isBST}(L) \land \forall x \in L.\, x < v \\ & \land \mathsf{isBST}(R) \land \forall x \in R.\, v < x\end{aligned}$$

Example 7 (Binary search in an array). Leon handles imperative programs through a source-to-source transformation into equivalent pure functional programs. We have show previously that this transformation preserves soundness of quantifier handling in this classic example of imperative program verification [16] (see Appendix B.2 for Leon source code). When presented with an underspecified version of this benchmark featuring the following incomplete loop invariant

```
0 ≤ low && low ≤ high + 1 && high < a.length &&
(if (res == −1) forall((i: Int) ⇒ (high + 1 ≤ i && i < a.length) ⟹ (a(i) != key))
else res ≥ 0 && res < a.length && a(res) == key))
```

the Leon framework can successfully report a counter-example to both the inductiveness of the invariant and the search soundness property.

Finally, quantifier support in Leon has been used in the context of other research to prove useful properties of programs with several hundred lines of source code. This contribution is outside the scope of this paper.

4 Related Work

Saturation-based theorem proving [1, 17] has been well-studied and underlies many semi-decision procedures for first-order logic with equality [6, 14]. Finite saturated clause sets can even lead to sound models in certain cases [13]. The restrictions imposed on our fragment clearly make our semi-decision procedure less general than these, however it provides meaningful insight into integration of saturation with E-matching based heuristics. Furthermore, as our procedure integrates with an SMT solver, ground clauses with theories can be soundly mixed with our first-order reasoning, as well as certain fragments of theories with quantifiers. Research in superposition with theories is indeed ongoing [2, 18].

E-matching has proved effective at handling quantification in SMT solvers [10, 11]. The technique provides strong theoretical results on certain restricted fragments [3] but lacks completeness guarantees in general. This limitation has been alleviated by using orthogonal approaches that use triggers to direct proof search but rely on theoretically stronger techniques for refutation-completeness [22]. We have shown that our model finding technique can also provide refutation-completeness for certain fragments without compromising on satisfiable cases.

In [12], an instantiation procedure is presented along with strong theoretical results, such as decidability in non-trivial fragments and semi-decidability for a large class of theories. The idea of exploring relevant domains until a fixpoint is obtained is quite similar to our own and can be extended with our model finding technique by slighly adapting the Δ_F system of equations. A more complete discussion of the relation between both techniques can be found in 2.3.

Given the nature of the models for which we have completeness, our technique obviously relates to finite model finding. As our approach avoids model representation, connections with methods for model enumeration [8, 23, 26] are fairly limited. Other techniques use range finiteness as basis for their procedure [4, 5, 9] and can elegantly handle finiteness of sort cardinalities [20, 21]. These techniques incrementally explore the relevant domain through increasing domain cardinality, whereas the semi-decision procedure directly underlies our search. Finally, in some cases, finite model finding techniques can be extended to domains with infinite cardinality [15].

5 Conclusion

We proposed a new instantiation-based semi-decision procedure for a fragment of first-order logic with equality that can use E-matching triggers as an instantiation heuristic. We presented a technique for model finding based on our semi-decision procedure and showed completeness for models where all functions have finite range. We further showed that our technique can be integrated into other procedures and can provide refutation-completeness for the considered fragment. We also discussed how our model finding technique can guide quantifier instantiations to increase their relevance. Furthermore, the instantiation procedure can be successfully applied to certain fragments of first-order logic with theories. Both the quantifier instantiation procedure and the model finding technique were integrated into a boader verification framework for pure higher-order functional programs. Interesting future research directions include extending model finding to support full first-order logic with equality, as well as considering more complex model validity criteria that can be encoded into the formula to broaden the scope of satisfying models our procedure can find.

References

1. L. Bachmair and H. Ganzinger. Equational reasoning in saturation-based theorem proving. *Automated deduction—a basis for applications*, 1:353–397, 1998.
2. L. Bachmair, H. Ganzinger, and U. Waldmann. Refutational theorem proving for hierarchic first-order theories. *AAECC*, 5:193–212, 1994.
3. K. Bansal, A. Reynolds, T. King, C. W. Barrett, and T. Wies. Deciding local theory extensions via E-matching. In D. Kroening and C. S. Pasareanu, editors, *CAV*, volume 9207 of *LNCS*, pages 87–105. Springer, 2015.
4. P. Baumgartner, J. Bax, and U. Waldmann. Finite quantification in hierarchic theorem proving. In S. Demri, D. Kapur, and C. Weidenbach, editors, *IJCAR*, volume 8562 of *LNCS*, pages 152–167. Springer, 2014.
5. P. Baumgartner, A. Fuchs, H. de Nivelle, and C. Tinelli. Computing finite models by reduction to function-free clause logic. *JAPLL*, 7(1):58–74, 2009.
6. P. Baumgartner and C. Tinelli. The model evolution calculus with equality. In R. Nieuwenhuis, editor, *CADE*, volume 3632 of *LNCS*, pages 392–408. Springer, 2005.
7. R. Blanc, V. Kuncak, E. Kneuss, and P. Suter. An overview of the Leon verification system: Verification by translation to recursive functions. In *4th Scala Workshop*, 2013.
8. J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *ITP*, 2010.
9. K. Claessen and N. Sörensson. New techniques that improve MACE-style finite model finding. In *Proceedings of the CADE-19 Workshop: Model Computation-Principles, Algorithms, Applications*, pages 11–27, 2003.
10. L. M. de Moura and N. Bjørner. Efficient e-matching for SMT solvers. In F. Pfenning, editor, *CADE*, volume 4603 of *LNCS*, pages 183–198. Springer, 2007.
11. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *JACM*, 52(3):365–473, 2005.
12. Y. Ge and L. M. de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *LNCS*, pages 306–320. Springer, 2009.
13. M. Horbach and C. Weidenbach. Decidability results for saturation-based model building. In R. A. Schmidt, editor, *CADE*, volume 5663 of *LNCS*, pages 404–420. Springer, 2009.
14. K. Korovin and C. Sticksel. iprover-eq: An instantiation-based theorem prover with equality. In J. Giesl and R. Hähnle, editors, *IJCAR*, volume 6173 of *LNCS*, pages 196–202. Springer, 2010.
15. V. Kuncak and D. Jackson. Relational analysis of algebraic datatypes. In M. Wermelinger and H. C. Gall, editors, *ESEC/SIGSOFT FSE*, pages 207–216. ACM, 2005.
16. K. R. M. Leino and R. Monahan. Dafny meets the verification benchmarks challenge. In G. T. Leavens, P. W. O'Hearn, and S. K. Rajamani, editors, *VSTTE*, volume 6217 of *LNCS*, pages 112–126. Springer, 2010.
17. R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. *Handbook of automated reasoning*, 1:371–443, 2001.
18. V. Prevosto and U. Waldmann. SPASS+T. *ESCoR*, 192:18–33, 2006.
19. A. Reynolds and V. Kuncak. Induction for SMT solvers. In D. D'Souza, A. Lal, and K. G. Larsen, editors, *VMCAI*, volume 8931 of *LNCS*, pages 80–98. Springer, 2015.

20. A. Reynolds, C. Tinelli, A. Goel, and S. Krstic. Finite model finding in SMT. In N. Sharygina and H. Veith, editors, *CAV*, volume 8044 of *LNCS*, pages 640–655. Springer, 2013.
21. A. Reynolds, C. Tinelli, A. Goel, S. Krstic, M. Deters, and C. Barrett. Quantifier instantiation techniques for finite model finding in SMT. In M. P. Bonacina, editor, *CADE*, volume 7898 of *LNCS*, pages 377–391. Springer, 2013.
22. P. Rümmer. E-matching with free variables. In N. Bjørner and A. Voronkov, editors, *LPAR*, volume 7180 of *LNCS*, pages 359–374. Springer, 2012.
23. J. K. Slaney. FINDER: finite domain enumerator - system description. In A. Bundy, editor, *CADE*, volume 814 of *LNCS*, pages 798–801. Springer, 1994.
24. P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *SAS*, 2011.
25. N. Voirol, E. Kneuss, and V. Kuncak. Counter-example complete verification for higher-order functions. In P. Haller and H. Miller, editors, *Scala@PLDI*, pages 18–29. ACM, 2015.
26. J. Zhang and H. Zhang. SEM: a system for enumerating models. In *IJCAI*, pages 298–303. Morgan Kaufmann, 1995.

A Proofs

Lemma 2. *If applying superposition to F obtains the empty clause, then there exists a sequence of selections by the* Progress *rule*

$$(b_1, f(\overline{s})_1) \in G_0 - I_0, \ldots, (b_n, f(\overline{s})_n) \in G_{n-1} - I_{n-1}$$

such that $I_n = \bigcup_{i=1}^{n} \{(b_i, f(\overline{s})_i)\}$ and $\bigcup_{i=0}^{n} E_i$ is unsatisfiable.

First, note that when unified terms are ground, the solver ensures validity of unification. Indeed, for t_1, t_2 ground, if $\mathrm{MGU}(t_1, t_2)$ exists, then $\models t_1 \simeq t_2$. This observation extends to ground literals l_1, l_2, such that $\mathrm{MGU}(l_1, l_2)$ existence implies $\models l_1 \iff l_2$. We therefore need only consider cases where one or both targets of unification contain quantified variables. As we are showing existence of a sequence of Progress rule selections, we can also assume that all quorums can be instantiated for any clause (instantiation feasibility simply requires extending the sequence). Moreover, one should note that as clauses, and therefore literals, are uniquely given by the function applications they contain, considering a literal l for superposition translates to its defining applications being in the set of future instantiations (as we can then assume the selection sequence contains them). Finally, variables in $\mathcal{V}$ are considered as constants in E_i, therefore identical variables across different clauses are *not* independent when considering satisfiability of $\bigcup_{i=1}^{n} E_i$.

Proof. We show that our procedure implies superposition by showing that the generated clauses imply valid unifications, and if the clause resulting from a superposition rule application is ground, then it is equisatisfiable with the clauses generated by our procedure (the empty clause is obviously ground). Let us consider the applications of each superposition rule.

RESOLUTION and EQUALITY. We start with RESOLUTION and consider three cases:

1. if only literal l_1 (respectively l_2) contains quantified variables, the INST rule will generate the clause $\neg c \vee \theta[\![l_L \vee l_1]\!]$, and $\sigma = \mathrm{MGU}(l_1, \neg l_2)$ existence implies that $\theta[\![l_L]\!] \vee l_D$ is ground, holds, and is equisatisfiable with $\sigma[\![l_L \vee l_D]\!]$,
2. if only corresponding portions of matched function application arguments in l_1 and l_2 are non-ground, INST will also enforce a valid unification as the non-ground portions are not (and need not be) unified,
3. otherwise, the UNIFY rule will introduce the literal l_2' such that unification of l_1 and l_2' falls into case 1 or 2.

The arguments for EQUALITY RESOLUTION are much the same where s and s' replace l_1 and $\neg l_2$. For EQUALITY FACTORING, the above reasoning tells us the clause $\neg c \vee \theta[\![l_L \vee s \simeq t \vee s \simeq t']\!]$ will be generated where c, θ correspond to $\sigma = \mathrm{MGU}(s, s')$ existence, and the underlying solver ensures that

$$M \models \neg c \vee \theta[\![l_L \vee s \simeq t \vee s \simeq t']\!] \iff M \models \neg c \vee \theta[\![l_L \vee t \not\simeq t' \vee s \simeq t]\!].$$

SUPERPOSITION. If $s' \notin \mathcal{V}$, then INST and UNIFY will ensure unification holds in much the same way as for RESOLUTION and EQUALITY. We can therefore assume without loss of generality that s' is $x \in \mathcal{V}$ and $s[u]$ is either ground, or independent from x and can be grounded by INST or UNIFY rule applications. Our definition of $\mathcal{G}$ ensures that the following clauses are generated

SUPERPOSITION RIGHT	SUPERPOSITION LEFT
$\{s' \to s[u]\}l_D \vee s[u] \simeq a$ and $a \not\simeq s[u]$	$\{s' \to s[u]\}l_D \vee s[u] \not\simeq s[u]$

In either case, the INST rule will then generate $\neg c \vee \theta[\![l_L \vee t \simeq t']\!]$. The existence of $\sigma = \mathrm{MGU}(t, u)$ ensures that $\theta[\![l_L \vee \{s' \to s[u]\}l_D \vee t \simeq t']\!]$ holds and is equisatisfiable with the clause resulting from the superposition rule when it is ground through similar arguments as when $s' \notin \mathcal{V}$.

If the empty clause results from superposition, then there must therefore exist a sequence of PROGRESS rule selections $(b_1, f(\overline{s})_1), \ldots, (b_n, f(\overline{s})_n)$ such that a subset $E \subseteq \bigcup_{i=0}^{n} E_i$ is unsatisfiable.

Lemma 3. *If for $(b, f(\overline{s}))$ in G_t, either $M \not\models b$ or $M[\![\overline{s}]\!] \in D_f$, then $M \models E_{t+1}$.*

Proof. We consider two independent cases:

$M \not\models b_i$ for at least one ground selection $(b_i, f(\overline{s})_i) \in I_{t+1}$ during instantiation implies that all generated clauses will be satisfied through $false \implies *$, and

$M[\![\overline{s}_i]\!] \in D_{f_i}$ for all ground selections $(b_i, f(\overline{s})_i) \in I_{t+1}$ ensures that there exist corresponding ground selections $(c_i, g(\overline{r})_i) \in I_t$ such that $f_i = g_i$, $M \models c_i$, and $M[\![\overline{r}_i]\!] = M[\![\overline{s}_i]\!]$. The INST inferrence rule further guarantees that the clauses resulting from the set $(c_i, g(\overline{r})_i)$ are in Σ_t and therefore all new clauses implied by $(b_i, f(\overline{s})_i)$ are satisfied by M.

B Source Code for Examples

B.1 Binary Search Tree

Example 6 (Binary Search Tree).

```
sealed abstract class Tree
case class Node(left: Tree, value: BigInt, right: Tree) extends Tree
case class Leaf() extends Tree

def content(tree: Tree): Set[BigInt] = tree match {
  case Leaf() ⇒ Set.empty[BigInt]
  case Node(l, v, r) ⇒ content(l) ++ Set(v) ++ content(r)
}

def isBST(tree: Tree) : Boolean = tree match {
  case Leaf() ⇒ true
  case Node(left, v, right) ⇒ {
    isBST(left) && isBST(right) &&
    forall((x:BigInt) ⇒ (content(left).contains(x)  ⟹  x < v)) &&
    forall((x:BigInt) ⇒ (content(right).contains(x)  ⟹  v < x))
  }
}

def insert(tree: Tree, value: BigInt): Node = {
  require(isBST(tree))
  tree match {
    case Leaf() ⇒ Node(Leaf(), value, Leaf())
    case Node(l, v, r) ⇒ if (v < value) Node(l, v, insert(r, value))
      else if (v > value) Node(insert(l, value), v, r)
      else Node(l, v, r)
  }
} ensuring (res ⇒ isBST(res) && content(res) == content(tree) ++ Set(value))
```

B.2 Binary Search in an Array

Example 7 (Binary search in an array).

```
def binarySearch(a: Array[BigInt], key: BigInt): Int = ({
  require(a.length > 0 && forall((i: Int, j: Int) ⇒
    (i ≥ 0 && j ≥ 0 && i < a.length && j < a.length && i < j) ⟹ (a(i) ≤ a(j))))

  var low = 0
  var high = a.length − 1
  var res = −1

  (while(low ≤ high && res == −1) {
    val o = if ((high & 1) == 1 && (low & 1) == 1) 1 else 0
    val i = high / 2 + low / 2 + o
    val v = a(i)

    if (v == key) res = i
    else if (v > key) high = i − 1
    else low = i + 1
  }) invariant (
    0 ≤ low && low ≤ high + 1 && high < a.length &&
    (if (res == −1)
      forall((i: Int) ⇒ (0 ≤ i && i < low) ⟹ (a(i) != key)) &&
      forall((i: Int) ⇒ (high + 1 ≤ i && i < a.length) ⟹ (a(i) != key))
    else
      res ≥ 0 && res < a.length && a(res) == key))
  res
}) ensuring(res ⇒ {
  if(res == −1) forall((i: Int) ⇒ (0 ≤ i && i < a.length) ⟹ (a(i) != key))
  else a(res) == key
})
```