A Model Guided Instantiation Heuristic for the Superposition Calculus with Theories

Joshua Bax

NICTA[*] and Australian National University, Canberra, Australia
Joshua.Bax@nicta.com.au

Abstract

Generalised Model Finding (GMF) is a quantifier instantiation heuristic for the superposition calculus in the presence of interpreted theories with finitely quantified free function symbols ranging into theory sorts. The free function symbols are approximated by finite partial function graphs along with simplifying assumptions which are iteratively refined. Here we present an outline of the GMF approach, give an improvement that addresses some efficiency issues and then present some ideas for extending it with concepts from instantiation based theorem proving.

1 Overview

The inclusion of interpreted theories in a theorem proving context naturally leads to completeness issues. In the classical first-order logic satisfiability problem we are at least guaranteed refutational completeness, but reasoning modulo theories and including uninterpreted function symbols is often not even semi-decidable. Many applications of theorem proving reduce to the satisfiability problem, from ranking function and loop invariant synthesis in software verification to counter-example finding. In addition to reasoning modulo theories, these applications also require the introduction of uninterpreted function symbols and quantifier reasoning.

We aim to recover some completeness in the specific case where there are only finitely many ground instances of (uninterpreted) terms ranging into the interpreted theory. This is guaranteed with the assumption that all variables inside uninterpreted terms of certain sorts are quantified over finite ranges.

In the following the interpreted theory will be Linear Integer Arithmetic (LIA), which we view as a particular class of models that satisfy the LIA axioms. Concretely, clauses over the LIA signature $(+^2, -^1, 0, s^1, <^2)$ are evaluated using Cooper's Quantifier Elimination algorithm which decides $\exists\forall$ quantified LIA formulas. Sorts, operators, terms and literals from the LIA theory are called *background*. New operators which extend the signature of the interpreted theory are called *foreground* or *free*.

Define *background-sorted foreground* (BSFG) terms as those which have at their head an uninterpreted function symbol with a background sort.

The running example will use the usual array theory with integer indices and elements, this has the operators $\mathsf{read} : \mathsf{Array} \times \mathbb{Z} \to \mathbb{Z}$, $\mathsf{write} : \mathsf{Array} \times \mathbb{Z} \times \mathbb{Z} \to \mathsf{Array}$ (these are foreground operators; read is a BSFG operator) and axioms $\mathcal{T}_{\mathsf{Array}} =$

$$(1)\quad \mathsf{read}(\mathsf{write}(A, I, X), I) \approx X \qquad (2)\quad \mathsf{read}(\mathsf{write}(A, I, X), J) \approx \mathsf{read}(A, J) \lor I \approx J$$

[*]NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

We adapt the Hierarchic Superposition [1, 5] framework for integrating superposition with background theories. This lays out conditions for refutational completeness of the combined procedures, namely *sufficient completeness* of the input clause set and *compactness* of the background theory.

Sufficient completeness requires that any ground instance of an uninterpreted term which has a background sort is provably equal to some background term in *all* first-order models of the ground instances of the input clause set (not just models extending the background theory).

Given a sufficiently complete clause set, a superposition procedure should eventually replace all uninterpreted theory sorted terms with background theory terms, and these can be used by the theory solver to perhaps refute the clause set. If this were not the case then it could be that the superposition procedure yields a saturated clause set (and therefore a model) which the theory solver cannot refute, but which does not extend any proper model of the theory. So even if the superposition procedure saturates a given clause set, without sufficient completeness we cannot conclude that it is satisfiable with respect to the chosen background theories.

Consider the read operator, since it has sort $\mathbb{Z}$ any clause set containing it and any ground instance $\mathsf{read}(\mathsf{a}, i)$ must be able to be proven equal to some integer. We may have a unit clause $X \leq \mathsf{read}(\mathsf{a}, i)$ say, which is satisfiable by some arbitrary model but not by any of the theory models. This is primarily a syntax issue; the theory solver cannot accept uninterpreted function symbols. If we also had $\exists Y. \mathsf{read}(\mathsf{a}, i) \approx Y$, this would allow rewriting the first clause to one involving just LIA symbols and a Skolem constant which can be refuted by the theory solver. If the BSFG term were not ground in the first clause this would be more difficult and in general, computing whether an arbitrary clause set has the sufficient completeness property is undecidable [13].

The background theory is compact if any unsatisfiable set of formulas over the background signature has a finite unsatisfiable subset. This is a requirement for refutational completeness because the solver for the interpreted theory can only accept finite sets of formulas. As a requirement of the method presented here we assume that the LIA signature is extended with countably many *parameters* (i.e. $\mathbb{Z}$-sorted constants) and fix the model class as those models of the LIA axioms which always interpret parameters as integers. If the class of models defining the background theory is the standard model of LIA augmented with all possible interpretations of the parameters, then this is a problem because the set $\{n \leq \alpha \mid n \in \mathbb{Z}\}$, where α is a parameter is unsatisfiable in every model but any finite subset is satisfiable in some model. Instead we assume that the background models are only those models which satisfy the first-order axioms of LIA, then the background models are compact by the general compactness of first-order logic. However the class of theory models now includes non-standard models of LIA, e.g. $\{n \leq \alpha \mid n \in \mathbb{Z}\}$ could be satisfied by a model $(\mathbb{Z}_2 \times \mathbb{Z})$ with a lexicographic ordering for $\leq$ and where the symbol n is interpreted as $(0, n)$ and α as $(1, 0)$. In the particular case here, we are guaranteed to not produce such infinite unsatisfiable sets by the assumption of finite quantifiers (effectively that there are finitely many background subterms). For more details on the particular configuration of Hierarchic Superposition used here, see [5]. For an in depth discussion of the issue of compactness in this context, see [4].

We describe the Generalised Model Finding algorithm, give some examples of its use and then present a refinement to the algorithm that eliminates some redundancies and finally present some directions for future research.

2 Generalised Model Finding

Generalised Model Finding (GMF)[2] is a generalisation of the 'define' calculus rule of [5] which replaces ground instances of BSFG terms as they are generated in the course of a derivation in order to recover some 'local' sufficient completeness. Here we assume finitely many ground instances of BSFG terms so that sufficient completeness can be effectively achieved by a similar replacement. This is generalised by assuming that these finite sets of BSFG terms are given succinctly by quantifying non-ground BSFG terms over finite sets. For example we might have $\forall X \in [0, 1000].\ \mathsf{read}(\mathsf{a}, X) \approx 5$. The objective is then to make these clauses sufficiently complete efficiently. This will be done using the superposition solver to prove or refute the clause set once it is made sufficiently complete.

Of course we could simply add all ground instances along with fresh parameters, but this will produce exponentially many instances. Instantiating in this way is similar to the approach taken by Finite Model Finders such as Paradox, Mace and Gandalf which also replace the terms with constants; although only implicitly by grounding of the input clause set. It is important to re-emphasise that here we are only grounding a subset of the uninterpreted subterms, others remain untouched. However, this restriction does not guarantee termination unlike Finite Model finding. What is guaranteed (with sufficient completeness) is the absence of false positives.

In order to obtain better scaling behavior, the GMF procedure initially assumes that the BSFG terms to be replaced in each clause are constant on the given domain: $\forall X \in [m, \dots, n].\ \mathsf{read}(\mathsf{a}, X) \approx \alpha$ where α is a fresh parameter. Likely this will be false, but there is a better chance that this can be detected by the superposition calculus now that sufficient completeness has been recovered. After demonstrating unsatisfiability we can refine the initial assumptions by finding a subset of clause instances which are satisfiable and a particular clause instance which makes this set unsatisfiable. We use this information to add exceptions to the assumption, e.g. our assumption could become $\forall X \in [m, \dots, n-1].\ \mathsf{read}(\mathsf{a}, X) \approx \alpha \wedge \mathsf{read}(\mathsf{a}, n) \approx \alpha_n$, where α_n is a new parameter not necessarily distinct from α. Eventually we will have added all possible exceptions and, if the clause set still remains unsatisfiable, then we have a proof of the unsatisfiability of the whole clause set.

On the other hand, if the clause set is satisfiable for some set of assumptions, then we have identified a model for the (finitely quantified) clause set.

On notation: variables will be upper-case X, Y, Z; parameters will be Greek letters α_i, β_i; concrete integers will be d_i; foreground operators will be in sans-serif $\mathsf{read}, \mathsf{write}, \mathsf{a}$; functionally the only difference between $\mathbb{Z}$-sorted foreground constants and parameters is that the foreground constants appear in the signature. Where variables appear without quantifiers they are assumed to be universally quantified.

The finite quantifiers will be encoded as part of the formula: so $\forall X \in \{d_0, \dots, d_n\}.\ \psi(X)$ becomes $\forall X.(X \approx d_0 \vee \ldots \vee X \approx d_n) \rightarrow \psi(X)$. When the finite set is a range of integers $[m, n]$, it could instead be $\forall X.(m \leq X \wedge X \leq n) \rightarrow \psi(X)$. The latter provides an alternative to explicitly enumerating all instances over a finite quantifier set, as these membership constraints can be dispatched by the LIA solver. For simplicity we assume here that all of these finite sets are sets of integers, though in principle they could be arbitrary pairwise distinct constants of any sort.

Here we will abstract the representation of these finite sets and simply represent them as $\forall X.\ S_X \rightarrow \psi(X)$, where S_X is a formula with free variable X which expresses membership in the appropriate finite set. For convenience S_X will represent both the set and the formula defining the set depending on context.

Let $\mathcal{FV}(C)$ be the finitely quantified variables of C. In general finitely quantified clauses are written $C \leftarrow S$, where S is a conjunction of formulas S_X for each $X \in \mathcal{FV}(C)$. Calling it a clause is a bit misleading since S can be an arbitrary formula and so $C \leftarrow S$ may be expanded to multiple clauses in a derivation.

As outlined above, we proceed by adding assumptions about BSFG terms in order to guarantee sufficient completeness. We now describe how to generate these assumptions for each clause.

For each of the clause variables we keep a set of exception points, which are initially empty. Our intent is to

1. Generate the set of definitions formed by replacing BSFG terms with a fresh constant at each of the relevant exception points;
2. Map any remaining points of the finite domain to a default constant, and
3. Simultaneously produce the result of replacing BSFG terms in instances of the clause using these definitions.

This is accomplished with the following procedure:

```
FD($C$,$Pi$):=
  Let $\mathrm{Cls}_C$:= {} and $\mathrm{Def}_C$:= {}
  for( every subset $V = \{Z_1,\ldots,Z_l\}$ of $\{X_1,\ldots,X_n\}$ ):
   for( every substitution $\gamma = [Z_1 \mapsto d_1,\ldots Z_l \mapsto d_l]$ where $d_m \in \Pi_{Z_m}$):
      Let $E := D\gamma$
      while ($E$ has the form $E[t]$ where $t$ is a minimal BSFG term):
        Let $\alpha$ be a fresh parameter
        Let $S_R$:= $\bigwedge_{X \notin V} (S_X \setminus \Pi_X)$
        add $t \approx \alpha \leftarrow S_R$ to $\mathrm{Def}_C$
        $E := E[\alpha]$
      add $E \leftarrow S_R$ to $\mathrm{Cls}_C$
  return ($\mathrm{Cls}_C$,$\mathrm{Def}_C$)
```

FD generalises to clause sets under the assumption that clauses do not share variables.

We call members of Def_C *definition clauses*; these have the general form $t[X] \approx \alpha \leftarrow S_X$ where t is a BSFG term, α is a parameter and S_X is a (formula specifying a) possibly empty finite set. Call the term t on the left of the positive equation a *definition term.*

The FD transform must be applied again to the original clause set after every modification to the set of exception points to ensure that assumptions are properly updated. This is done in the following loop, using the procedure `find` to generate new exception points. Given a clause set CS:

```
GMF($CS$):=
  except:={}
  while(true):
    $CS_F$:=FD($CS$,except)
    if ($CS_F$ is SAT): return ‘model found’
    else if ($CS_F \setminus$ finitelyQuantified($CS_F$) is UNSAT):
      return UNSAT //no refinements possible
    else:
      $(X,d)$ := find($CS_F$)
      except += $(X,d)$
```

The function `finitelyQuantified` returns the subset of clauses with non-empty finite quantifiers. Satisfiability tests are carried out by passing the current set of clauses to the superposition solver. The function `find` is a heuristic that identifies the next exception point to add using the superposition solver to test for unsatisfiability. Since adding an exception point only increases the possible models that the definitions allow we could even just return an arbitrary pair (X, d). In the paper the search procedure is roughly:

1. For each clause C and finitely quantified variable X of C, cumulatively remove definitions and instances of C which depend on X until the clause set becomes satisfiable.
2. Find a subset S of S_X such that substituting S for S_X in remaining instances and definitions of C makes the clauses satisfiable but doing the same with $S \cup \{d\}$ produces unsatisfiability. This is done by a binary search on S_X (as a set) and is the source of most of the computational overhead of this procedure; it requires $O(\log(|S_X|))$ prover calls.
3. Return (X, d) (or an arbitrary one if the search is unsuccessful).

Example 2.1. *Consider a simple example: establishing the existence of an array* a *whose values are sorted in increasing order and for which at least two consecutive values are strictly greater. Define the clause set* N *as:*

$$\begin{array}{llll} (1) & \mathsf{read}(\mathsf{write}(A, I, X), I) \approx X & (4) & 1 \leq m \wedge m < 1000 \\ (2) & \mathsf{read}(\mathsf{write}(A, I, X), J) \approx \mathsf{read}(A, J) \vee I \approx J & (5) & \mathsf{read}(\mathsf{a}, m) < \mathsf{read}(\mathsf{a}, m+1) \\ (3) & \multicolumn{3}{l}{\mathsf{read}(\mathsf{a}, I) \leq \mathsf{read}(\mathsf{a}, J) \vee \neg(I < J) \leftarrow I \in [1..1000] \wedge J \in [1..1000]} \end{array}$$

In the following $I \in [1..1000]$ will be abbreviated to S_I and similarly for J. Notice that m is a foreground $\mathbb{Z}$-sorted constant, but will be treated no differently than a parameter.

Apart from simple renaming definitions for (5): $\alpha_1 < \alpha_2$, $\mathsf{read}(\mathsf{a}, m) \approx \alpha_1$, $\mathsf{read}(\mathsf{a}, m+1) \approx \alpha_2$ (which will be (5a)-(5c)), the finite domain transform is only necessary for (3) and produces:

$$\begin{array}{llll} (3a) & \multicolumn{3}{l}{\alpha_3 \leq \alpha_4 \vee \neg(I < J) \leftarrow S_I \wedge S_J} \\ (3b) & \mathsf{read}(\mathsf{a}, I) \approx \alpha_3 \leftarrow S_I & (3c) & \mathsf{read}(\mathsf{a}, J) \approx \alpha_4 \leftarrow S_J \end{array}$$

Here Cls_3 is (3a) and $\mathrm{Def}_3 = \{(3b), (3c)\}$. The new clause set $N_1 = \{(1), (2), (3a) - (3c), (4), (5a) - (5c)\}$ now needs to be checked for satisfiability. Because N_1 is sufficiently complete and hierarchic superposition decides the underlying fragment, we get a definite result and deduce that N_1 is in fact unsatisfiable.

Now, notice that removing clauses (3a)-(3c) gives a satisfiable set, but adding either (3b) or (3c) also yields a satisfiable clause set. This corresponds to the find procedure identifying (3) and I as a possible target for refinement. Next subsets of S_I will be tested by directly substituting for S_I and retrying the satisfiability test. The point identified by our procedure will be the number 1000 for the variable I. That is, replacing S_I by $I \in [0..999]$ in N_1 gives an unsatisfiable set again. Then $\mathrm{FD}((3), (I \mapsto \{1000\}))$ is

$$\begin{array}{llll} (3a1) & \multicolumn{3}{l}{\alpha_{31} \leq \alpha_4 \vee \neg(I < J) \leftarrow S_I \setminus \{1000\} \wedge S_J} \\ (3a2) & \multicolumn{3}{l}{\alpha_{32} \leq \alpha_4 \vee \neg(1000 < J) \leftarrow S_J} \\ (3b1) & \mathsf{read}(\mathsf{a}, I) \approx \alpha_{31} \leftarrow S_I \setminus \{1000\} & (3c) & \mathsf{read}(\mathsf{a}, J) \approx \alpha_4 \leftarrow S_J \\ (3b2) & \mathsf{read}(\mathsf{a}, 1000) \approx \alpha_{32} & & \end{array}$$

Clauses (3b1) and (3b2) provide the modified definitions and clauses (3a1) and (3a2) are the correspondingly rewritten versions of (3). Let $N_3 = \{(1), (2), (3a1) - (3c), (4), (5a) - (5c)\}$ be the result of the current transformation step.

The clause set N_3 is still unsatisfiable. In the next iteration the exception point $J \mapsto 1000$ is identified in (3). Let $N_4 = \{(1), (2), (4), (5a) - (5c)\} \cup \mathrm{FD}((3), (I \mapsto \{1000\}, J \mapsto \{1000\}))$. This time, N_4 is satisfiable, and hence so is N with the same models. If $\mathcal{I}$ is any such model we have $\mathcal{I}(m) = 999$, $\mathcal{I}(\mathsf{read}(\mathsf{a}, I)) = k$, for some integer k and all $I = 1..999$, and $\mathcal{I}(\mathsf{read}(\mathsf{a}, 1000)) = l$ for some integer $l > k$.

The whole example is solved after two iterations of transformation steps. With $m = 2$ (the number of finitely quantified variables) and $n = 1000$ this accounts for $2 \cdot (m \cdot \log(n)) \leq 40$ theorem prover calls, each of which is rather trivial. By contrast, the full ground instantiation of the clauses (3)-(5) has a size of $n^m = 10^6$ which, in general, grows too quickly for current theorem provers or SMT solvers.

Example 2.2. *Consider the example of an ordering given by the predicate P where all variables range over $[0, 1, 2]$ embedded as part of a larger clause set (4) ...*

$$\begin{array}{ll}
(1) & P(X, X) \\
(2) & \neg P(X, Y) \vee \neg P(Y, X) \vee X \approx Y \\
(3) & \neg P(X, Y) \vee \neg P(Y, Z) \vee P(X, Z) \\
(4) & \ldots
\end{array}$$

Note that we assume (1)-(3) are embedded as part of a larger clause set, since superposition already saturates these axioms. For any variable X Let $S_X := X \in [0, 1, 2]$. The purpose of this example is simply to illustrate certain shortcomings of the default method. To use GMF we treat P as a function to the domain $\{true, false\}$. Then the initial FD transform simply replaces the predicates with Boolean variables:

$$\begin{array}{llll}
(1a) & \alpha_1 \approx true & (1b) & \alpha_1 \approx P(X, X) \leftarrow S_X \\
(2a) & \alpha_2 \not\approx true \vee \alpha_3 \not\approx true \vee X \approx Y \leftarrow S_X \wedge S_Y & (2b) & \alpha_2 \approx P(X, Y) \leftarrow S_X \wedge S_Y \\
 & & (2c) & \alpha_3 \approx P(Y, X) \leftarrow S_X \wedge S_Y \\
(3a) & \alpha_4 \not\approx true \vee \alpha_5 \not\approx true \vee \alpha_6 \approx true & (3b) & \alpha_4 \approx P(X, Y) \leftarrow S_X \wedge S_Y \\
(3c) & \alpha_5 \approx P(Y, Z) \leftarrow S_Y \wedge S_Z & (3d) & \alpha_6 \approx P(X, Z) \leftarrow S_X \wedge S_Z
\end{array}$$

The left hand term of all definitions are *variants*, terms which unify with a substitution σ which is a bijection from variables to variables. So this first approximation essentially assumes P is constant everywhere and, as there are both positive and negative instances of P in the clause set, this is unsatisfiable.

Next is the repair phase. Binary search is not able to identify any satisfiable subsets so all we can do is add the exception $X = 0$ for (2) arbitrarily.

Then FD((2), $(X \mapsto \{0\})$ contains the new definition $\alpha_7 \approx P(0, Y) \leftarrow S_Y$ and instance $\alpha_7 \not\approx true \vee c_3 \not\approx true \vee 0 \approx Y \leftarrow S_Y$ as well as the original definitions updated with $S_X = [1, 2]$. However the overall clause set remains unsatisfiable as definitions (3b)-(3d) still contain $P(X, Y)$ with $S_X, S_Y = [0, 1, 2]$ and so force the interpretation of P to be everywhere constant again.

This does not change until the exception point $X \mapsto \{0\}$ is added in (3) as well. But then the same sequence of redundant updates happens again for the variants of $P(0, Y), P(1, Y)$ and so on.

So there is considerable duplication of work as a result of keeping multiple variants of literals. The procedure would benefit from having some centralised means for storing and applying definitions, and for updating them in a way that takes into account the logical structure of the current definitions. We address the issue of redundant variants in the next section.

3 Variant aware GMF

As observed above, keeping definitions for multiple variants of a BSFG term leads to inefficiencies in GMF. To address this we will develop a modification of the GMF procedure in which definitions are stored in common for all clauses in order to eliminate redundancies. Exception points should only need to be added once for any particular term. Also this will hopefully allow the algorithm to align more closely with the idea of repairing *interpretations* of BSFG terms rather than a syntax dependent replacement operation.

Instead of explicitly generating definition clauses as in the original approach, now assumptions will be stored in a separate datastructure.

Definition 3.1 (Assumption). *An* assumption *is a tuple* (α, t, S) *where* α *is a parameter,* t *is a BSFG term and* S *is a finite set of substitutions* σ *such that* $t\sigma$ *is ground.*

As before, the substitution set S is also a formula over the variables of the definition term t. If the formula defining S is $\psi_S[X_1, \ldots, X_n]$ then the equivalence is $S = \{\sigma : \text{for all } X_i,\ X_i\sigma \in \mathbb{Z} \text{ and } \psi_S\sigma \equiv \top\}$. Assumptions can also be viewed simply as the set of ground equations $\{t\sigma \approx \alpha : \sigma \in S\}$. It follows from the definition that S is empty if and only if t is already ground.

In this section we make use of the idea of *simple substitutions*. This is any substitution σ such that for all background sorted variables X, $X\sigma$ is either a variable, a member of the background theory domain (i.e. $\mathbb{Z}$), or a parameter. Since we assume that all finite quantifier sets are over subsets of the integers, each substitution in the substitution set of an assumption is simple.

Given a clause set $\mathcal{C}$ we produce an assumption set $\mathcal{A}$. The definition terms of an assumption set are all those terms that occur in $\mathcal{A}$. In keeping with our goal we require that no variant definition terms exist in $\mathcal{A}$. Then assumptions within $\mathcal{A}$ can be indexed either by parameter or by definition term, as both of these are unique.

Now that assumptions are decoupled from clauses we need a way to produce the set of instances of $\mathcal{C}$ with respect to the current assumptions. Before, this was done simultaneously in the FD transform and a reference to the original clause was kept along with the new instances.

Here we will make use of matchers to identify the assumptions to apply to a particular occurrence of a BSFG term in a clause. A term s *matches* term t with matcher μ if μ is a simple substitution such that $s\mu = t$.

In order to simplify the retrieval of relevant assumptions, the initial assumptions are produced by taking *maximal* BSFG subterms (i.e. those that are not contained in any BSFG term) from clauses in $\mathcal{C}$. So we are guaranteed that at least (variants of) the maximal BSFG terms are amongst the definition terms.

The initial step which produces the set of assumptions we will call `Gen`. It consists of taking maximal BSFG terms, adding these as new assumptions and merging the substitution sets of any assumptions with variant definition terms.

```
Gen($\mathcal{C}$):=
  Let $\mathcal{A}$:= {}
  for ($C \leftarrow S$ in $\mathcal{C}$):
    if (($\alpha$, $s$, $S_s$) $\in$ $\mathcal{A}$ such that $s \sim t$ with matcher $\mu$):
      update ($\alpha$, $s$, $S_s$) in $\mathcal{A}$ to ($\alpha, s\mu, (S_s \cup S)\mu$)
    else: $\mathcal{A} := \mathcal{A} \cup \{(\alpha, t, S_t)\}$
  return $\mathcal{A}$
```

The application of a substitution μ to a substitution set S has the same effect as applying μ to the equivalent formula and building a new substitution set; formally, $(S)\mu = \{\sigma \in S : \sigma = \mu{\cdot}\eta\}$.

We can characterise updates as substitutions applied to the set of substitutions in assumptions: for example, given the assumption $(\alpha, \mathsf{f}(\mathsf{g}(Y), X), X \in [0,2] \wedge Y \in [0,3])$ an update might identify the term $\mathsf{g}(Y)$ and exception point $Y \mapsto 1$. Then the updated assumptions will be

$$(\alpha, \mathsf{f}(\mathsf{g}(Y), X), X \in [0,2] \wedge Y \in [0,3] \wedge Y \not\approx 1)$$
$$(\alpha_1, \mathsf{f}(\mathsf{g}(1), X), X \in [0,2])$$

In general, given an assumption (α, t, S_t) and a non-empty substitution set S, `Update`$((\alpha, t, S_t)$,$S)$ is $\{(\alpha, t, S_t \setminus S)\} \cup \{(\alpha_\sigma, t\sigma, S_t\sigma) : \sigma \in S\}$ where each α_σ is fresh.

To relate this back to the original clause(s) that contains subterm $\mathsf{f}(\mathsf{g}(Y), X)$, we can build a (simple) matcher. All modified assumptions can then be found by finding all definition terms which are variants or have matchers with the original clause term. So the instance of the clause $C[s] \leftarrow S$ produced by an assumption (α, t, S_t), given that $s\mu = t$ for simple matcher μ, is $(C[\alpha] \leftarrow (S \wedge S_t))\mu$.

This is formalised in the `Apply` procedure which produces clause instances relative to a given assumption set. We will define this in a moment as it depends on the notion of labels, to be introduced.

So we can produce clause instances after an update- we still need to find this update! The binary search heuristic of the original GMF algorithm was based on a complex indexing of clauses and the variables they contained as well as the instances and definitions derived from them. Here we don't have this complex index structure and need some way to track the assumptions used in producing a clause instance or a definition clause.

Recall that clause instances are produced by matching clause terms with definition terms.

Definition 3.2 (Labelled Clause)**.** *A labelled clause $C \mid \mathcal{L}$ is a clause C and a set of assumptions $\mathcal{L} = \{A_1, \ldots, A_n\}$.*

The idea is that the label of C contains all assumptions that are necessary to produce it, both in the initial `Apply` step and in a derivation. The substitution sets in the label assumptions will accumulate any substitutions applied to C in the course of a derivation. Definition clauses derived in the sufficient completeness transform are also labelled by the assumption they originate from. (This idea has much in common with Labelled superposition [14] and Constrained Resolution [7]).

The inference rules of the superposition calculus can be simply modified to propagate labels using the following scheme:

$$\frac{C_1 \mid \mathcal{L}_1 \ \ldots \ C_n \mid \mathcal{L}_n}{D \mid (\bigcup_{1 \le i \le n} \mathcal{L}_i)\sigma}$$

where the labelled clauses $C_i \mid \mathcal{L}_i$ are premises of some inference rule, σ is the unifier used in the inference and D is the conclusion of the inference.

When a labelled empty clause is derived the labels will encode the assumptions necessary for the derivation of that particular result. To repair a derivation given a label $\{A_1, \ldots, A_n\}$, it is enough to add exceptions for substitutions in just one assumption label. So if we choose $A_i = (\alpha_i, t_i, S_i)$, then we modify the original assumption (α_i, t_i, S) by replacing it with the set of assumptions `Update`$((\alpha_i, t_i, S)$,$S_i)$, removing variants if necessary. If an empty clause with an empty label is derived then the clause set is unsatisfiable as a whole, as the derivation of the contradiction is independent of any assumptions. Similarly if all assumptions have empty substitution sets, as the set of assumptions cannot be modified to avoid the contradiction.

Here is the method Apply which produces instances of a clause under an assumption set by forming all possible matchings between the maximal BSFG terms in the clause and multisets of assumptions. Let $C[t_1,\dots,t_n] \leftarrow S$ be a clause with finite quantifiers encoded by S, maximal BSFG terms t_i and $\mathcal{A}$ be an assumption set.

```
Apply($C \leftarrow S$,$\mathcal{A}$):
  Let $C_{\mathcal{A}}$:= {}
  for (all combinations of assumptions $A_1,\dots,A_n$):
    assume $A_i = (\alpha_i, s_i, S_i)$
    if(exists $\mu$ such that $t_i\mu$ = $s_i$ where $s_i$ is a fresh variant for all $i$):
      add $C[\alpha_1,\dots,\alpha_n]\mu \leftarrow (S \wedge \bigwedge_i S_i)\mu$ | $\{A_1\mu \dots A_n\mu\}$ to $C_{\mathcal{A}}$
  return $C_{\mathcal{A}}$
```

Note that in building the matcher an assumption might occur multiple times, but each occurrence must be a fresh variant. This method needs to be called on every change to the assumption set in order to produce any new clause instances that may arise.

In order to use the Hierarchic superposition calculus at all we must produce a sufficiently complete clause set. The instantiated clause set is sufficiently complete since it contains no BSFG terms at all. The assumptions need to be transformed into sufficiently complete clauses, this is more involved because the definition terms can contain nested BSFG terms. The procedure to do this is similar to FD in the original GMF procedure:

```
Flatten($A$):=
  assume $A = (\alpha, t, S_t)$
  Let $E$:=$t$ and $\mathrm{Def}_t$:= {}
  while($E$ has the form $E[s]$ where $s$ is a minimal BSFG term):
    Let $\beta$ be a fresh parameter
    $\mathrm{Def}_t$:= $\mathrm{Def}_t \cup \{s \approx \beta \leftarrow S_t$ | $A\}$
    $E$:= $E[\beta]$
  return $\mathrm{Def}_t \cup \{E \approx \alpha \leftarrow S_t$ | $A\}$
```

If $\mathcal{A}$ is an assumption set then Flatten($\mathcal{A}$) is the result of applying Flatten to all assumptions in $\mathcal{A}$.

Overall the new algorithm will take a finitely quantified clause set $\mathcal{C}$ and proceed as follows:

```
GMF-var($\mathcal{C}$):=
  $\mathcal{A}$:= Gen($\mathcal{C}$)
  while(true):
    if( $\square$ | $\mathcal{L}$ is in Saturate(Apply($\mathcal{C}$,$\mathcal{A}$) $\cup$ Flatten($\mathcal{A}$)):
      if($\mathcal{L}$ = {} or all $A \in \mathcal{L}$ are ground): return Unsat
      else:
        take non-ground $(\alpha, t, S) \in \mathcal{L}$
        Let $\mathcal{A}$:= Update(A$_\alpha$,$S$)
    else: return Sat
```

The method Saturate applies the modified labelling superposition calculus until a labelled empty clause is derived or no calculus rules are applicable, and then returns the derived clauses.

Example 3.1. *Consider example 2.1 again. Clauses (1), (2) and (4) are as above*

$$
\begin{array}{ll}
(3) & \mathsf{read}(\mathsf{a}, I) \leq \mathsf{read}(\mathsf{a}, J) \vee \neg(I < J) \leftarrow I \in [1..1000], J \in [1..1000] \\
(5) & \mathsf{read}(\mathsf{a}, m) < \mathsf{read}(\mathsf{a}, m+1)
\end{array}
$$

First produce the initial assumption set, removing variants: Gen(N) $=$

$$
\begin{array}{llll}
(A_1) & (\alpha_1, \mathsf{read}(\mathsf{a}, m), \{\}) & (A_2) & (\alpha_2, \mathsf{read}(\mathsf{a}, m+1), \{\}) \\
(A_3) & (\alpha_3, \mathsf{read}(\mathsf{a}, K), \{[K \mapsto d] : d \in [1..1000]\}) & &
\end{array}
$$

Notice that only one assumption A_3, results from the two maximal BSFG terms $\mathsf{read}(\mathsf{a}, I)$, and $\mathsf{read}(\mathsf{a}, J)$ in (3). The substitution set of A_3 is just the representation of S_I, S_J as a set of substitutions and will be abbreviated as S_K.

Next we produce the sufficiently complete, rewritten and labelled version of clauses and definitions:

$$\begin{array}{llll}
(3a) & \alpha_3 \leq \alpha_3 \vee \neg(K < K) \leftarrow S_K \mid A_3 & & \\
(3b) & \mathsf{read}(\mathsf{a}, K) = \alpha_3 \leftarrow S_K \mid A_3 & & \\
(5a) & \alpha_1 < \alpha_2 \mid A_1, A_2 & & \\
(5b) & \mathsf{read}(\mathsf{a}, m) = \alpha_1 \mid A_1 & (5c) & \mathsf{read}(\mathsf{a}, m+1) = \alpha_2 \mid A_2
\end{array}$$

Clause (3a) is trivially true. The proof simply rewrites the LHS of (5b) and (5c) with (3b) to get

$$\alpha_3 \approx \alpha_1 \leftarrow 1 \leq m \leq 1000 \mid A_1, A_3[K \mapsto m]$$
$$\alpha_3 \approx \alpha_2 \leftarrow 1 \leq m+1 \leq 1000 \mid A_2, A_3[K \mapsto m+1]$$

These both rewrite (5a) to give $\alpha_3 < \alpha_3 \leftarrow 1 \leq m \leq 1000 \wedge 1 \leq m+1 \leq 1000 \mid A_1, A_2, A_3[K \mapsto m], A_3[K \mapsto m+1]$; then this clause set is closed using the background solver. The label for the empty clause is $A_1, A_2, A_3[K \mapsto m], A_3[K \mapsto m+1]$

Already we see a critical difficulty- what should be done with the parameter m in the final assumption set? In ideal conditions, we would arrive at a set of concrete exceptions for an assumption, if $m = 1000$ initially then that would be the case. We can't add new instances of the definition terms $\mathsf{read}(\mathsf{a}, m)$, $\mathsf{read}(\mathsf{a}, m+1)$ as these already exist in the current assumption set. The assumptions A_1 and A_2 can't be modified as their substitution sets are empty.

Perhaps some concrete value of K could be guessed as an exception, say $K \mapsto 1000$, and in this case that would produce a satisfiable set. But this is not optimal; here only two exception points 0 and 1000 will give a satisfiable set.

Another option is to try a binary search on the domain identified by the assumption set and offending substitution. Here it is S_K in A_3, so we could restart the proof with $S_K = K \in [1..500]$ or $S_K = K \in [501..1000]$ and continually split the set S_K until we find a set $S \subset S_K$ and concrete value $d \in S_K$ such that replacing the formula S_K with the formula for $S \cup \{d\}$ in the rewritten clause set produces unsatisfiability, but substituting S for S_K is satisfiable.

Example 3.2. *Consider the previous example 2.2 and the first iteration of* `GMF` *given there.*

`Gen`(N) produces

$$(A_1) \quad (\alpha_1, P(X,X), S_X) \qquad (A_2) \quad (\alpha_2, P(X,Y), S_X \wedge S_Y)$$

This still enforces that P is constant everywhere, however it contains far fewer definitions and parameters.

$$\begin{array}{rl}
(1) & \alpha_1 \mid A_1 \\
(2a1) & \neg\alpha_1 \vee \neg\alpha_1 \vee X \approx X \leftarrow S_X \mid A_1 \\
(2a2) & \neg\alpha_2 \vee \neg\alpha_2 \vee X \approx Y \leftarrow S_X \wedge S_Y \mid A_2 \\
(3a) & \neg\alpha_1 \vee \alpha_2 \mid A_1, A_2 \\
(3b) & \neg\alpha_2 \vee \alpha_1 \mid A_1, A_2 \\
(4) & \ldots
\end{array}$$

The clause (2a1) is trivial, there are no matchers of clause (2) and A_1, A_2 together. Also any result of matching just A_2 or A_1 with (3) is trivial as it contains $\alpha_2 \wedge \alpha_2$. (3a) is formed by matching $P(X,Y), P(Y,Z), P(X,Z)$ with the (fresh variants of) definition terms $P(X_2,Y_2), P(X_3,Y_3), P(X_1,X_1)$. Similarly for (3b). The matchers for both of these do not refer to any specific values, so the substitution sets in the labels for (3a) and (3b) are unchanged.

Since (3a) and (3b) are complementary, we derive a labelled empty clause with label $\{A_1, A_2\}$. Then the substitution sets require adding *all* instances of either A_1 or A_2 as exceptions.

We might select A_1 as this will produce the least number of new instances (recall $S_X = [0,1,2]$)

$$\begin{array}{llll} (A_3) & (\alpha_3, P(0,0),) & (A_4) & (\alpha_4, P(1,1),) \\ (A_5) & (\alpha_4, P(2,2),) & & \end{array}$$

Assumption A_1 no longer appears.

The next iteration still yields an unsatisfiable clause set due to A_2 forcing α_3, α_4 and α_5 to be identical, however the empty clause will now have a label in which A_2 is specialised. For example if A_3 has been used in a similar way to A_1 was used in the previous step this would produce a label like $A_2[X \mapsto 0]$. This in turn allows adding the exception $A_6 = (\alpha_6, P(0,Y), S_y)$.

This was the stage we left the original example at. Recall that this required first a binary search on (the transformed) (2), each step of which required a call to the superposition solver, followed by another binary search to produce the same exception from (3). This still scales with domain size (even if it is only logarithmically). With this new method we have achieved all that in only two steps, and this is independent of the domain size entirely.

4 Related Work

Related work for the overall GMF approach can be found in [2]. Here we give work related to the specific modification in the last section.

Satisfiability Modulo Theories (SMT) solvers deal with the same problems in regard to quantifier reasoning over background theories. Since SMT solvers are strongest when reasoning over ground formulas, the general approach is to produce instances of the formula in question and use this to build a candidate model. This model can be checked against the quantified part of the original formula. This is essentially the Model Based Quantifier Instantiation idea described in [10], which also gives syntactic restrictions that guarantee complete instantiation. Also Reynolds et al. [15] give instantiation techniques for dealing with finite models. The key difference is that with GMF we are allowed free usage of quantification over non-BSFG terms and strictly background formulas, with the cost of non-termination in certain satisfiable cases.

Bonacina et al. [6] add hypothetical inferences to a combination SMT/Superposition proof procedure and use the backtracking mechanism of SMT solvers to repair any falsified assumptions. They give a general framework for including unsound conclusions in a derivation and mark these in a similar way to the label scheme here. It would be interesting to see whether the problem of efficiently replacing BSFG terms could also fit into that context.

As mentioned before, the class of Instance based methods also maintain a set of assertions about the clause set which are iteratively updated and can produce models for first-order clause sets in certain cases. The Inst-Gen calculus [9] uses unification to produce a set of possibly conflicting instances to pass to a Sat solver (or SMT solver). In a relevant variation, the Inst-Gen-Eq calculus [14] uses an SMT solver to select literals of the clause set which may combine to form a model, then a version of unit superposition with labels is used to extract contradictory instances that follow from this candidate model.

The Model-Evolution calculus [3] maintains a set of literals to represent the current candidate model of the clause set and also uses unification to produce possibly conflicting clauses in order to refine this model. More on this in the next section.

In contrast the GMF technique only operates over BSFG terms, essentially it attempts to produce an interpretation for these. Although GMF does not attempt to solve the same problems as Instance based methods it can use similar ideas for storing and updating an interpretation upon derivation of conflicts as in Model-Evolution or InstGen.

5 Future Work

Compare this with the Model Evolution (ME) calculus [3] which, like propositional DPLL, maintains a set of asserted literals (the context) and conducts a proof search by unifying input clauses against the current context in order to find potentially falsifying instances which are then added to the context by a backtracking split rule. Analogies can be made between the definition context and the ME context; also between the refinement procedure and the instantiation by unification. The difference is that the context carries more 'semantic weight' than the set of function definitions since it encodes a smaller number of models (essentially only those satisfying the literals it contains) versus all models which identify particular subterms as in the case of GMF. The definition refinement process is limited in that it only considers single variables at a time and each update involves the entire clause set. The context-unification approach considers a single clause at a time which is either made true by the current context or an instance of it is split on, after which it is no longer considered.

In addition certain theories perform poorly in equational reasoning contexts, the foremost example is finite domain theories [11] (e.g. which include axioms $\forall X.\ X = \mathbf{c}_1 \vee \ldots \vee X = \mathbf{c}_n$). Unfortunately clauses like these appear frequently in software verification contexts, usually as the encoding of particular finite types, e.g. `boolean` or a user defined enumerated type. Merging the GMF and ME procedures would lead to the possibility of encoding these theories in a form which they are 'hidden' from the equational proof procedure entirely.

Implementation We have an implementation for the original GMF version available in the latest release of beagle[1]. This implementation was used to generate the results in [2]. We plan on extending this with the ideas discussed here and testing in a similar way on a larger set of benchmarks.

Using existing definitions Sufficient completeness is undecidable in general, but certain clauses can express sufficient completeness quite simply, for example $\mathsf{f}(X) \approx X + 1$. Then, supposing we can identify a subset of clauses which give sufficient completeness how can we exploit them in a principled way? Already if we just consider clauses of the above form, i.e. unit equations which are flat on the left hand side and only contain background operators on the right side then it seems possible to simply include them in the definition context as is and pre-emptively rewrite using them whenever another instance of f is encountered. Could this be generalised say to Horn clauses where the positive literal has the unit equation form above and where all the negative literals are from the background theory? This would encode a definition plus a domain. For example $\mathsf{f}(X) \approx X \vee \neg(X > 0)$; the fact that the domain is not finite is no problem since we can use the positive equation to give sufficient completeness where possible.

[1]http://users.cecs.anu.edu.au/ baumgart/systems/beagle

Template functions A related approach would be to allow the user to specify a set of templates which can be used to confer sufficient completeness. For example you could specify $X+b$, where b is a parameter, as a template for single variable operators. This idea is mentioned in the context of model finding with SMT solvers in [8] and for finding loop invariants also using a quantifier elimination procedure in [12]. Again integrating this approach would require a principled combination with any method for adding sufficient completeness for function symbols not covered by the templates.

References

[1] Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. Refutational theorem proving for hierachic first-order theories. *Appl. Algebra Eng. Commun. Comput*, 5:193–212, 1994.

[2] Peter Baumgartner, Joshua Bax, and Uwe Waldmann. Finite quantification in hierarchic theorem proving. In S. Demri, D. Kapur, and C. Weidenbach, editors, *IJCAR 2014*, volume 8562 of *LNAI*, pages 152–167, Vienna, 2014. Springer Switzerland. To appear.

[3] Peter Baumgartner, Björn Pelzer, and Cesare Tinelli. Model evolution with equality – revised and implemented. *Journal of Symbolic Computation*, 47(9):1011–1045, September 2012.

[4] Peter Baumgartner and Uwe Waldmann. Hierarchic superposition: Completeness without compactness. In Marek Kosta and Thomas Sturm, editors, *MACIS 2013 –Fifth International Conference on Mathematical Aspects of Computer and Information Sciences*, pages 8–12, 2013.

[5] Peter Baumgartner and Uwe Waldmann. Hierarchic superposition with weak abstraction. In Maria Paola Bonacina, editor, *CADE-24 – The 24th International Conference on Automated Deduction*, volume 7898 of *Lecture Notes in Artificial Intelligence*, pages 39–57. Springer, 2013.

[6] Maria Paola Bonacina, Christopher Lynch, and Leonardo Mendonça de Moura. On deciding satisfiability by theorem proving with speculative inferences. *J. Autom. Reasoning*, 47(2):161–189, 2011.

[7] Hans-Jürgen Bürckert. A resolution principle for constrained logics. *aij*, 66(2):235–271, 1994.

[8] Leonardo De Moura and Nikolaj Bjørner. Bugs, moles and skeletons: Symbolic reasoning for software development. In *Automated Reasoning*, pages 400–411. Springer, 2010.

[9] H. Ganzinger and K. Korovin. Integrating equational reasoning into instantiation-based theorem proving. In *Computer Science Logic (CSL'04)*, volume 3210 of *Lecture Notes in Computer Science*, pages 71–84. Springer, 2004.

[10] Yeting Ge and Leonardo Mendonça de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2009.

[11] Thomas Hillenbrand and Christoph Weidenbach. Superposition for finite domains. Research Report MPI-I-2007-RG1-002, Max-Planck Institute for Informatics, Saarbruecken, Germany, April 2007.

[12] Deepak Kapur. A quantifier-elimination based heuristic for automatically generating inductive assertions for programs. *Journal of Systems Science and Complexity*, 19(3):307–330, 2006.

[13] Deepak Kapur, Paliath Narendran, Daniel J Rosenkrantz, and Hantao Zhang. Sufficient-completeness, ground-reducibility and their complexity. *Acta Informatica*, 28(4):311–350, 1991.

[14] Konstantin Korovin and Christoph Sticksel. Labelled unit superposition calculi for instantiation-based reasoning. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 459–473. Springer, 2010.

[15] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstić, Morgan Deters, and Clark Barrett. Quantifier instantiation techniques for finite model finding in SMT. In M. P. Bonacina, editor, *Proceedings of the 24th International Conference on Automated Deduction (Lake Placid, NY, USA)*, volume 7898 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2013.