Choose Your Colour: Tree Interpolation for Quantified Formulas in SMT

Elisabeth Henkel[1(✉)], Jochen Hoenicke[2], and Tanja Schindler[3]

[1] University of Freiburg, Freiburg im Breisgau, Germany
henkele@informatik.uni-freiburg.de
[2] Certora, Tel Aviv-Yafo, Israel
jochen@certora.com
[3] University of Liège, Liège, Belgium
tanja.schindler@uliege.be

Abstract. We present a generic tree-interpolation algorithm in the SMT context with quantifiers. The algorithm takes a proof of unsatisfiability using resolution and quantifier instantiation and computes interpolants (which may contain quantifiers). Arbitrary SMT theories are supported, as long as each theory itself supports tree interpolation for its lemmas. In particular, we show this for the theory combination of equality with uninterpreted functions and linear arithmetic. The interpolants can be tweaked by virtually assigning each literal in the proof to interpolation partitions (colouring the literals) in arbitrary ways. The algorithm is implemented in SMTInterpol.

Keywords: Tree Interpolation · Quantified Formulas · SMT

1 Introduction

Craig interpolants [7] were originally proposed to reason about proof complexity. In the last two decades, research reignited when interpolants proved useful for software verification, in particular for generating invariants [15]. Tree interpolants are useful for verifying programs with recursion [12], and for solving non-linear Horn-clause constraints [23], which can be used for thread modular reasoning [10,16] and verifying array programs [20]. For many verification problems, reasoning about first-order quantified formulas is needed. Quantified formulas are, among others, needed to model unsupported theories or to express global properties of arrays [19], for example, sortedness [3,24].

An interpolation problem is an unsatisfiable conjunction of several input formulas, the partitions of the interpolation problem. An interpolant summarises the contribution of a single or multiple partitions to the unsatisfiability. Interpolants can be computed from resolution proofs. However, most methods require localised proofs where each literal is associated with some input partition [22]. Proofs generated by SMT solvers, especially with quantifier instantiations, usually contain mixed terms and literals created during the solving process that cannot be associated with a single input formula.

© The Author(s) 2023
B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 248–265, 2023.
https://doi.org/10.1007/978-3-031-38499-8_15

In this paper, we extend our work on proof tree preserving sequence interpolation of quantified formulas [13]. The method presented therein allows for the computation of inductive sequence interpolants from instantiation-based resolution proofs of quantified formulas in the theory of uninterpreted functions. The key idea of this method is to perform a virtual modification of mixed terms introduced through quantifier instantiations, thus allowing to compute an inductive sequence of interpolants on a single, non-local proof tree.

We extend the interpolation algorithm to compute tree interpolants and to support arbitrary SMT theories (with the single restriction that such a theory itself must support tree interpolation for its lemmas). We simplify the treatment of mixed terms by virtually flattening all literals independently of the partitioning. We show that the literals can be coloured (assigned to a partition) arbitrarily, and that for every colouring, correct interpolants are produced. The interpolants contain quantifiers for the flattening variables that bridge different partitions, and by choosing colours sensibly the number of quantifiers can be reduced. In contrast to previous works [1,12] which produce tree interpolants by repeated binary interpolation and require multiple proofs, our method computes a tree interpolant from a single proof.

Related Work. Many practical algorithms to compute interpolants have been presented. We focus here on proof-based methods that either work in the presence of quantifiers, or that can compute tree interpolants, or both.

Our work builds on the method presented in [4] for computing interpolants from instantiation-based proofs in SMT. It is based on *purifying* quantifier instantiations by introducing variables for terms not fitting the partition, and adding defining auxiliary equalities as a new input clause in the proof. Our method introduces these variables and equalities only virtually for computing the partial interpolants. Thus, tree interpolants can be computed from a single proof of unsatisfiability, while in [4] a purified proof is required for each partition.

There exist several methods to compute interpolants for quantified formulas inductively from superposition-based proofs. In [2], each literal is given a *label* (similar to our colouring) used to project the clause to the different partitions. First, a *provisional* interpolant is computed that may contain local symbols. These symbols are replaced by quantified variables to obtain an interpolant. In contrast to our method, the approach only works when the provisional interpolants contain at most local constants, i.e., no local functions or predicates, and the assignment of labels is not flexible as our colouring. The method in [17] is based on a slightly modified proof, where substitution steps are done separately. First, a *relational* interpolant is computed, which may contain local function symbols, but only shared predicates. In logic without equality, or when the only local symbols are constants, the relational interpolant can be turned into an interpolant by quantifying over non-shared terms, respecting their dependencies.

A very different method based on summarising subproofs is presented in [9]. The proof is split into subproofs belonging to a single partition. The relevant subproofs are summarised in an *intermediant* stating that their premises imply

their conclusion. If the subproofs contain only symbols of the respective partition, the resulting formula is an interpolant. If the proof can be split in that way, the method works for any theory and proof system, but for tree interpolation, a different proof would be required for each partitioning.

Tree interpolants can be computed by repeated binary interpolation from formulas where the children interpolants are included, as discussed in [12]. In the propositional setting, [11] discusses under which conditions sets of interpolants with certain relations, such as tree interpolants, can be obtained by binary interpolation on different partitionings of the same formula. The method is implemented in OpenSMT, but the solver, and therefore the interpolation engine, does not support quantifiers.

A general framework for computing tree interpolants for ground formulas from a single proof has been presented in [5]. It works for combinations of equality-interpolating theories and is based on projecting *mixed literals* using auxiliary variables and predicates. Additionally, the rule for computing a resolvent's interpolant from its antecedents' interpolants is more involved. The method cannot deal with quantifier instantiations, nor with terms mixing subterms from different partitions. We discuss in Sect. 6 how it can be combined with the interpolation method for quantified formulas presented in this paper.

The first implementation of a tree interpolation algorithm in the presence of quantifiers and theories was in Vampire [1]. It is based on repeatedly computing binary interpolants for modified interpolation problems, similar to [12]. For each binary computation, the proof must be localised in order to be able to compute interpolants. In contrast, our method computes tree interpolants in one go from a single proof that has been obtained without knowledge of the partitioning of the tree interpolation problem. To the best of our knowledge, Vampire is the only other tool that is able to compute tree interpolants in the presence of quantifiers.

2 Notation

We assume that the reader is familiar with first-order logic. We define a *theory* $\mathcal{T}$ by its *signature*, that contains constant, function and predicate symbols, and its set of *axioms*, closed formulas that fix the meaning of those function and predicate symbols that are *interpreted* by the theory.

A *term* is a variable or the application of an n-ary function symbol to n terms. An *atom* is the application of an n-ary predicate to n terms, and a *literal* is an atom or its negation. A *clause* is a disjunction of literals, and a formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses. We use $\top$ (resp. $\bot$) for the formula that is always true (resp. false).

We will demonstrate our algorithm using the theory of equality, and the theory of linear arithmetic (with rationals and/or integers). The theory of equality establishes reflexivity, symmetry, and transitivity of the equality predicate $=$, and congruence for each *uninterpreted* function symbol. For simplicity of the presentation, uninterpreted constants are considered as 0-ary functions, and uninterpreted predicate symbols as uninterpreted functions with Boolean return

value. The theory of linear arithmetic contains the predicates $\leq, <$, rational constants c, the binary addition function $+$, and a family of unary multiplication functions $c\cdot$, one for each rational constant c. These symbols have their usual semantics, and the main theory lemmas are trichotomy $(x < y \lor x = y \lor x > y)$ and a variant of Farkas lemma. For simplicity, we apply arithmetic conversions implicitly and treat $x \leq y$ and $y \geq x$ and $1 \cdot x + (-1) \cdot y \leq 0$ as the same literal, and $x > y$ as its negated literal.

We denote constants by a, b, c, functions by f, g, h, variables by v, x, y, z, and terms by s, t. We use ℓ for literals, C for clauses, and ϕ, F, I for formulas.

For a term t, the outermost (or *head*) function symbol is denoted by $hd(t)$. The set of all uninterpreted function symbols occurring in a formula F is $symb(F)$ and the set of all free variables in F is $FreeVars(F)$. The result of substituting in a formula F each occurrence of a variable x by a term t is denoted by $F\{x \mapsto t\}$. By $\bar{x}$ and $\bar{t}$, we denote the list of variables $x_1, \ldots, x_n$ and terms $t_1, \ldots t_n$, respectively. We use the symbol $\equiv$ to denote equivalence between formulas, and to assign a formula to a formula variable.

3 Preliminaries

Craig Interpolation. A binary *Craig interpolant* [7] for an unsatisfiable conjunction $A \land B$ is a formula I that is implied by A, contradicts B, and contains only symbols that occur in both A and B. A generalisation are tree interpolants, which introduce several partitions in a tree-like structure.

Definition 1 (Tree interpolation). *A* tree interpolation problem (V, E, F) *is a labelled binary tree where V is a set of nodes connected by directed edges $E \subseteq V \times V$ pointing towards the root node. Every node except for the root node has one outgoing edge to its parent, and each non-leaf node has exactly two incoming edges. The* partitions $P \subseteq V$ *of the tree interpolation problem are the leaf nodes. The labelling function F assigns a formula to each partition $p \in P$ of the tree such that their conjunction is unsatisfiable. We use $st(v) \subseteq P$ to denote the set of leaves in the subtree of the node v, i.e., the set of leaves for which a path to the node v exists.*

A tree interpolant *for the interpolation problem (V, E, F) is a labelling function I for all nodes with the following properties:*

1. *The label $I(v_r)$ of the root node v_r of the tree is $\bot$.*
2. *For each leaf node $p \in P$, its interpolant $I(p)$ is implied by the formula $F(p)$.*
3. *For each inner node $v \in V \setminus P$, its interpolant $I(v)$ is implied by the conjunction $I(v_1) \land I(v_2)$ of the interpolants labelled to the two child nodes v_1, v_2.*
4. *For each node v, the symbols in $I(v)$ occur both inside and outside the subtree $st(v)$, i.e., $symb(I(v)) \subseteq (\bigcup_{p \in st(v)} symb(F(p))) \cap (\bigcup_{p \notin st(v)} symb(F(p)))$.*

Remarks. In contrast to the earlier definition of tree interpolation [1,5], only the leaves of the tree are labelled by F here. A tree interpolation problem with labelled inner nodes can be transformed to our formalism by adding a leaf child to each such node. A non-binary tree can be extended to a binary tree by adding more internal nodes. If the interpolants of the newly created nodes are ignored, the remaining interpolants are tree interpolants according to the earlier definition for tree interpolation.

A binary interpolant of A and B corresponds to the tree interpolant of the tree containing just two leaves A and B, more precisely, it is the interpolant labelled to the first leaf. Vice versa, each interpolant $I(v)$ of a tree interpolant is also a binary interpolant of the formulas in the partitions $A := st(v)$ and $A^c := P \setminus st(v)$. Since the set A defines v uniquely, we can also use I_A to denote $I(v)$. We call a symbol A-*local* if it only occurs in partitions in A, A^c-*local* if it only occurs in partitions in A^c, and *shared* if it occurs in both. The interpolant may only contain shared symbols.

Theory Combination. We assume that the solver uses Nelson–Oppen style theory combination sharing equalities without explicitly introducing auxiliary variables, and that each lemma in the proof belongs to one theory. Subterms in these lemmas containing symbols from a different theory are treated as if they were auxiliary variables. We further assume that there is a theory-specific interpolation procedure for the lemmas. In this paper, we do not have the assumption that theories are equality-interpolating. We introduce quantifiers in the interpolants for such theories. However, our approach can also be combined with equality-interpolating theories and corresponding procedures to avoid quantifiers, see Sect. 6.

CNF Transformation and Quantifiers. We assume that complex input formulas are transformed to CNF by Tseitin-encoding, which introduces Boolean proxy atoms. Existentially quantified variables are replaced with Skolem constants or functions (if nested under a universal quantifier) and conjunctions are lifted over universal quantifiers. Complex subformulas under a universal quantifier are replaced by uninterpreted predicates, taking as arguments the quantified variables. Quantified Tseitin-style axioms give the meaning for these predicates. Thus, we end up with quantified clauses of the form $\forall \bar{x}.\, \ell_1(\bar{x}) \vee \cdots \vee \ell_n(\bar{x})$, which we treat as a proxy literal. Instances of quantified clauses are created using instantiation lemmas of the form $\neg(\forall \bar{x}.\, \ell_1(\bar{x}) \vee \cdots \vee \ell_n(\bar{x})), \ell_1(\bar{t}), \ldots, \ell_n(\bar{t})$ where $\bar{t}$ are ground terms. Note that the proxy atom for a quantified formula occurs only positively in input clauses and negated in instantiation lemmas. We note that all preprocessing steps are done locally for each input formula, and that auxiliary predicates and Skolem functions are fresh predicate or function symbols. An interpolant of the preprocessed formulas is also an interpolant of the original formulas, because the auxiliary symbols are not shared between different input formulas and will never appear in the interpolant.

Proofs. A *resolution proof* for the unsatisfiability of a formula in CNF is a derivation of the empty clause $\bot$ using the resolution rule

$$\frac{C_1 \vee \ell \qquad C_2 \vee \neg\ell}{C_1 \vee C_2}$$

where C_1 and C_2 are clauses, and ℓ is a literal called the *pivot* (literal). A resolution proof can be represented by a tree, or more generally, if the same subproof is used more than once, by a directed acyclic graph (DAG). In our setting, the DAG has three types of leaves: *input clauses*, *theory lemmas*, i.e., clauses that are valid in the theory $\mathcal{T}$, and *instantiation lemmas* of the form $\neg(\forall\bar{x}.\phi(\bar{x})) \vee \phi(\bar{t})$. The inner nodes are clauses obtained by resolution, and the unique root node is the empty clause $\bot$.

Binary interpolants can be computed from a resolution proof by computing so-called partial interpolants for each clause. Each proof step proves a clause C as a consequence of the input $A \wedge B$, hence it proves that $A \wedge B \wedge \neg C$ is unsatisfiable. If each literal in the proof is assigned to, or *coloured* with, either partition A or B, a *partial interpolant* for each intermediate step is the interpolant of $A \wedge \neg C \downharpoonright A$ and $B \wedge \neg C \downharpoonright B$, where the projection $\neg C \downharpoonright A$ extracts from the conjunction $\neg C$ all literals that are coloured with partition A. McMillan showed for propositional logic that partial interpolants (cf. Definition 2 in [18]) can be computed recursively for each resolution step as the disjunction of the partial interpolants of the antecedents if the pivot is coloured as A, and their conjunction if it is coloured as B.

4 Colouring of Terms and Literals

In this section, we fix an interpolation problem (V, E, F), with partitions $P \subseteq V$. We use the following example to illustrate our interpolation algorithm.

Example 1 (Running example). Take the tree interpolation problem with nodes $V = \{123, 1, 23, 2, 3\}$ and edges $E = \{(1, 123), (23, 123), (2, 23), (3, 23)\}$ (see also Fig. 1), where the partitions $P = \{1, 2, 3\}$ are labelled with $F(p) \equiv \phi_p$ where

$$\phi_1 \equiv \forall x.\ g(h(x)) \leq x, \quad \phi_2 \equiv \forall y.\ g(y) \geq b, \quad \phi_3 \equiv \forall z.\ f(g(z)) \neq f(b).$$

The conjunction of the three formulas is unsatisfiable. Instantiating ϕ_1 with b gives $g(h(b)) \leq b$. Instantiating ϕ_2 with $h(b)$ gives $g(h(b)) \geq b$. Together they imply $g(h(b)) = b$. However, this contradicts ϕ_3 instantiated with $h(b)$. This proof creates, among others, the new literal $g(h(b)) \leq b$. The term $g(h(b))$ contains function symbols that do not occur in a common partition.

We recall that by $symb(F(p))$, we denote the uninterpreted function symbols occurring in the formula $F(p)$. We also keep track of the partitions where a symbol occurs:

Definition 2 (Partitions). *The* partitions of a function symbol f *are the partitions where this symbol occurs:*

$$partitions(f) = \{p \in P \mid f \in symb(F(p))\}.$$

McMillan's interpolation algorithm assumes that all symbols of a literal occur in one partition, such that the literal can be coloured with that partition. This is no longer the case in SMT, because new literals are created during the proof search, especially in the presence of instantiation lemmas. Our solution to this problem is to split each literal into many smaller literals and assign each of them to a partition. To keep the presentation simple, we flatten all (non-proxy) literals using a fresh variable for each application term. Thus, for every term t occurring in the resolution proof, we create a fresh variable v_t and associate with it a set of flattening equalities. In each literal, the top-level terms are replaced with their associated variable, and the defining equalities are conjoined.

Definition 3 (Flattening). *For a term t, we introduce a fresh variable v_t, and similarly for all its subterms. The associated set of flattening equalities $FlatEQ(t)$ is defined as follows:*

$$FlatEQ(t) = \{v_{f(t_1,\ldots,t_n)} = f(v_{t_1},\ldots,v_{t_n}) \mid f(t_1,\ldots,t_n) \text{ is a subterm of } t\}.$$

The flattened version of a literal ℓ is

$$flatten(\ell) \equiv \begin{cases} v_{t_1} = v_{t_2} & \text{if } \ell \equiv t_1 = t_2 \\ c_1 \cdot v_{t_1} + \cdots + c_n \cdot v_{t_n} \leq c & \text{if } \ell \equiv c_1 \cdot t_1 + \cdots + c_n \cdot t_n \leq c \end{cases}$$

and the associated set of flattening equalities is as follows

$$FlatEQ(\ell) = \begin{cases} FlatEQ(t_1) \cup FlatEQ(t_2) & \text{if } \ell \equiv t_1 = t_2 \\ FlatEQ(t_1) \cup \cdots \cup FlatEQ(t_n) & \text{if } \ell \equiv c_1 \cdot t_1 + \cdots + c_n \cdot t_n \leq c. \end{cases}$$

The flattened version of a negated literal is the negation of the flattened literal, i.e., $flatten(\neg\ell) \equiv \neg flatten(\ell)$. The set of flattening equalities for a negated literal is the set of flattening equalities for the literal itself, i.e., $FlatEQ(\neg\ell) = FlatEQ(\ell)$.

The conjunction of the equalities in $FlatEQ(t)$ implies that $v_t = t$. Similarly, the conjunction $flatten(\ell) \wedge \bigwedge FlatEQ(\ell)$ implies the literal ℓ and is equisatisfiable to ℓ. Proxy literals like quantified formulas are not flattened, as they will never occur in a partial interpolant. For such a proxy literal, $flatten(\forall x.\phi(x)) \equiv \forall x.\phi(x)$ and $FlatEQ(\forall x.\phi(x)) = \emptyset$.

Example 2 (Flattening). Consider the literal $g(h(b)) \leq b$. Its flattened version is $flatten(g(h(b)) \leq b) \equiv v_{g(h(b))} \leq v_b$, and the set of flattening equalities is

$$\begin{aligned} FlatEQ(g(h(b)) \leq b) &= FlatEQ(g(h(b))) \cup FlatEQ(b) \\ &= \{v_{g(h(b))} = g(v_{h(b)}), v_{h(b)} = h(v_b), v_b = b\}. \end{aligned}$$

To define partial interpolants, we colour each atom ℓ with some partition, denoted by $colour(\ell) \in P$. The negated atom always has the same colour. For proxy atoms created during the CNF conversion, it is important to colour them

with the input partition from which they were created. The colour of other literals can be chosen arbitrarily, but a good heuristic would choose a partition where most of the outermost function symbols occur. Each flattening equality is associated with all partitions where the corresponding function symbol occurs. The *projection* of auxiliary equations on a partition p, denoted by $\mathit{FlatEQ}(\ell) \downharpoonright p$, is defined as the conjunction of the equalities $(v_{f(t_1,\dots,t_n)} = f(v_{t_1},\dots,v_{t_n})) \in \mathit{FlatEQ}(\ell)$ where $p \in \mathit{partitions}(f)$.

Finally, we define the projection of a literal ℓ to a partition p. The *projection kernel* $\ell \downharpoonright^- p$ is $\mathit{flatten}(\ell)$ if $p = \mathit{colour}(\ell)$ or $\top$ otherwise. The *projection of* ℓ *to* p is defined as $\ell \downharpoonright p \equiv \ell \downharpoonright^- p \wedge \mathit{FlatEQ}(\ell) \downharpoonright p$. We define the projection to a set of partitions $\ell \downharpoonright A$ with $A \subseteq P$ (and similarly $\ell \downharpoonright^- A$) as the conjunction of all projections $\ell \downharpoonright p$ with $p \in A$. For a conjunction of literals $F \equiv \ell_1 \wedge \dots \wedge \ell_n$, we define $F \downharpoonright p \equiv \ell_1 \downharpoonright p \wedge \dots \wedge \ell_n \downharpoonright p$ and similar for $F \downharpoonright A$, $F \downharpoonright^- p$ and $F \downharpoonright^- A$.

Example 3 (Projection of literals). Consider again the literal $g(h(b)) \leq b$ from our running example (Example 1), and assume that we arbitrarily assign it to partition 2, i.e., $\mathit{colour}(g(h(b)) \leq b) = 2$. We have $\mathit{partitions}(g) = \{1,2,3\}$, $\mathit{partitions}(h) = \{1\}$ and $\mathit{partitions}(b) = \{2,3\}$. The projections are hence:

$$
\begin{aligned}
g(h(b)) \leq b \downharpoonright 1 &\equiv v_{g(h(b))} = g(v_{h(b)}) \wedge v_{h(b)} = h(v_b)\\
g(h(b)) \leq b \downharpoonright 2 &\equiv v_{g(h(b))} \leq v_b \wedge v_{g(h(b))} = g(v_{h(b)}) \wedge v_b = b\\
g(h(b)) \leq b \downharpoonright 3 &\equiv v_{g(h(b))} = g(v_{h(b)}) \wedge v_b = b
\end{aligned}
$$

Similar to the last paragraph in Sect. 3, we define a partial interpolant of a clause C as an interpolant of the input problem and $\neg C$. More precisely, it is the tree interpolant of a slightly modified tree interpolation problem, where the projection $\neg C \downharpoonright p$ is added to each leaf node $p \in P$. Since this step adds flattening variables potentially shared between several partitions, these variables can occur in the interpolants. The following definition accounts for the variables occurring in the projection of a clause.

Definition 4 (Supported variable). *We call a variable* v_t supported by a clause C *if its corresponding term* t *is a subterm of a non-proxy literal* ℓ *in* C.

The partial tree interpolant of a clause C may then contain a variable v_t as long as it is supported by the clause C.

Definition 5 (Partial tree interpolant). *A* partial tree interpolant *for a clause* C *is a tree interpolant as defined in Definition 1 for the tree interpolation problem* (V, E, F') *where the leaves are labelled with* $F'(p) \equiv F(p) \wedge \neg C \downharpoonright p$. *For the symbol condition, all variables supported by the clause may occur in all partial interpolants.*

5 Interpolation for Quantified Formulas

In the following, we describe how to compute tree interpolants for instantiation-based resolution proofs. We assume that each literal has been assigned to exactly

one partition of the tree interpolation problem, as described in the previous section. Following McMillan's algorithm, we compute partial tree interpolants inductively over the proof tree. The leaves of the proof tree are theory lemmas, for which we use theory-specific interpolation procedures, or they are input clauses or instantiation lemmas, for which we compute partial tree interpolants as described below. The inner nodes are obtained by resolution steps, for which we follow McMillan's algorithm to combine interpolants, and additionally treat variables that violate the symbol condition, as described later in this section.

5.1 Interpolation Algorithm

We start by explaining how the interpolants for leaf nodes are computed. Our algorithm computes interpolants separately for each node $v \in V$ in the tree interpolation problem. As mentioned in the preliminaries, we set $A = st(v)$ and use I_A to denote the interpolant $I(v)$.

Input Clauses. We assume that each input clause occurs in exactly one partition. The partial tree interpolant for an input clause C from partition p is given by $I_A \equiv \neg(\neg C \downharpoonleft^- A^c)$ if $p \in A$, and $I_A \equiv \neg C \downharpoonleft^- A$ if $p \notin A$.

Note that the literals can be assigned to a different partition than the clause. Although it makes sense to assign a literal to the same partition as the input clause it occurs in, this is not possible when the literal occurs in several input clauses. Therefore, the above formulas are not necessarily $\top$ or $\bot$. Proxy literals always have the same colour as the input clause and will therefore never appear in the interpolant.

Instantiation Lemmas. The partial tree interpolant for an instantiation lemma C obtained from a quantified input clause $\forall x.\phi(x)$ from partition $colour(\forall x.\phi(x))$ is computed in the same way as for input clauses.

Theory Lemmas. We only require that for each theory one can compute a partial tree interpolant for its lemmas, or to be more precise, the flattened negated lemmas. Thus, we can reuse any existing procedure. For self-containment, we cover transitivity, congruence, trichotomy and Farkas lemmas, which are the kind of lemmas our solver produces for the theory of equality and linear arithmetic.[1]

For a *transitivity* lemma with the corresponding conflict $\neg C \equiv t_1 = t_2 \wedge \cdots \wedge t_{n-1} = t_n \wedge t_1 \neq t_n$ we can ignore the auxiliary equations introduced by flattening the terms, as the projection kernel is also a transitivity lemma. A partial tree interpolant is computed by summarising for each A the chains of the flattened equalities (and, if applicable, the single disequality) that are assigned to a partition $p \in A$. More precisely, let $i_1 < \cdots < i_m$ be the boundary indices such that $colour(t_{i_j-1} = t_{i_j}) \in A$ and $colour(t_{i_j} = t_{i_j+1}) \notin A$ or vice versa. Set $i_1 = 1$ if $t_1 \neq t_n$ and $t_1 = t_2$ are in different partitions and $i_m = n$ if $t_{n-1} = t_n$

[1] Branches in linear integer arithmetic [8] are decisions on inequality literals and are handled by our resolution rule.

and $t_1 \neq t_n$ are in different partitions. If $m = 0$, then all colours of the equalities are in A and the interpolant is $\bot$, or they are all in A^c and the interpolant is $\top$. Otherwise, the interpolant summarises the equalities between the boundary indices that have a colour in A: if $colour(t_1 = t_n) \notin A$, then the interpolant is $I_A \equiv v_{i_1} = v_{i_2} \wedge v_{i_3} = v_{i_4} \wedge \cdots \wedge v_{i_{m-1}} = v_{i_m}$, otherwise the interpolant is $I_A \equiv v_{i_2} = v_{i_3} \wedge \cdots \wedge v_{i_{m-2}} = v_{i_{m-1}} \wedge v_{i_m} \neq v_{i_1}$. Here, v_i denotes the auxiliary variable introduced for t_i.

The flattened version of the conflict corresponding to a *congruence* lemma $C \equiv f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n) \vee t_1 \neq s_1 \vee \cdots \vee t_n \neq s_n$ is

$$\begin{aligned}
&v_{f(t_1,\ldots,t_n)} \neq v_{f(s_1,\ldots,s_n)} \wedge v_{t_1} = v_{s_1} \wedge \ldots \wedge v_{t_n} = v_{s_n} \\
&\wedge v_{f(t_1,\ldots,f_n)} = f(v_{t_1}, \ldots, v_{t_n}) \wedge v_{f(s_1,\ldots,s_n)} = f(v_{s_1}, \ldots, v_{s_n}) \\
&\wedge \bigwedge\{\ell \mid \ell \in \mathit{FlatEQ}(t), t \in \{t_1, \ldots, t_n, s_1, \ldots, s_n\}\}.
\end{aligned}$$

Note that the formula is still a congruence conflict if we drop the last line. Consequently, the flattening equalities for the arguments of the f-applications, and for their subterms, are not needed in the computation of a partial interpolant, they only establish the implication between the flattened and the original lemma. To obtain a partial tree interpolant, we first choose an arbitrary partition $p_f \in \mathit{partitions}(f)$. The partial tree interpolant is computed as follows.

$$I_A \equiv \begin{cases} \neg(\neg C \downharpoonleft^- A^c) & \text{if } p_f \in A \\ \neg C \downharpoonleft^- A & \text{otherwise} \end{cases}$$

For a *trichotomy* lemma $C \equiv t_1 = t_2 \vee t_1 > t_2 \vee t_1 < t_2$, both $I_A \equiv \neg C \downharpoonleft^- A$ and $I'_A \equiv \neg(\neg C \downharpoonleft^- A^c)$ are partial interpolants. We can always choose the projection that contains at most one literal.

A *Farkas* lemma has the form $C \equiv \neg(s_1 \leq b_1) \vee \cdots \vee \neg(s_n \leq b_n)$ where s_i is of the form $c_{i1} \cdot v_1 + \ldots + c_{im} \cdot v_m$ and b_i, c_{ij} are numeric (integer) constants. It is a valid lemma if there are Farkas coefficients (numeric integer constants) $k_1, \ldots, k_n > 0$ with $\sum_{i=1}^n k_i \cdot s_i = 0$ and $\sum_{i=1}^n k_i \cdot b_i < 0$. We assume that the lemma is flattened and all v_i are variables. The flattening equalities can be omitted from the lemma without changing its validity. For a set of partitions A, we denote by $L_A := \{i \mid colour(s_i \leq b_i) \in A\}$ the indices where $s_i \leq b_i$ is A-local. The partial tree interpolant for a Farkas lemma is computed by summing up the A-local literals multiplied by their Farkas coefficients. We obtain $I_A \equiv (\sum_{i \in L_A} k_i \cdot s_i) \leq (\sum_{i \in L_A} k_i \cdot b_i)$. Variables whose coefficients sum to zero are removed from the inequality. If A contains all inequalities, they sum up to the conflict $0 \leq \sum_{i=1}^n k_i \cdot b_i$ and we set $I_A \equiv \bot$.

Theorem 1. *The interpolants as defined in this section are valid partial tree interpolants for the respective leaf nodes.*

The proof for this theorem is a straight-forward case distinction over the type of leaf node. Details can be found in [14].

Resolution Steps. In a resolution step, we obtain the partial interpolant of the resolvent using the partial interpolants of the premises.

$$\frac{C_1 \vee \ell : I_A^1 \qquad C_2 \vee \neg\ell : I_A^2}{C_1 \vee C_2 : I_A^3}$$

As the first step, we follow McMillan's algorithm and combine the interpolants of the premises either with $\vee$ or with $\wedge$ depending on whether the pivot literal is A or A^c-local. For tree interpolants, this is done separately for each node of the tree interpolation problem, and a literal is seen as A-local if its colour is one of the leaves in the subtree of the node.

$$I_A^3 \equiv \begin{cases} I_A^1 \vee I_A^2 & \text{if } \mathit{colour}(\ell) \in A \\ I_A^1 \wedge I_A^2 & \text{if } \mathit{colour}(\ell) \notin A \end{cases}$$

The formula I_A^3 computed above may still contain variables supported by the antecedents that are no longer supported by the resolvent $C_1 \vee C_2$. Each of those *unsupported* variables must either be replaced by its definition or bound by a quantifier in the partial tree interpolant. More precisely, let v_t be an unsupported variable such that t is not a subterm of t' with $v_{t'} \in \mathit{FreeVars}(I_A^3)$. This variable must always exist, as there is always an outermost unsupported variable. Let $t = f(t_1, \dots, t_n)$. We replace I_A^3 as follows:

$$I_A^3 \equiv \begin{cases} \exists x.\, I_A^3\{v_t \mapsto x\} & \text{if } f \text{ is } A\text{-local, i.e., } \mathit{partitions}(f) \subseteq A, \\ \forall x.\, I_A^3\{v_t \mapsto x\} & \text{if } f \text{ is } A^c\text{-local, i.e., } \mathit{partitions}(f) \cap A = \emptyset, \\ I_A^3\{v_t \mapsto f(v_{t_1}, \dots, v_{t_n})\} & \text{if } f \text{ is shared (otherwise).} \end{cases}$$

We do this repeatedly for all variables in $\mathit{FreeVars}(I_A^3)$ that are unsupported. The variables may be treated in any order that respects the partial order induced by the subterm relation as described above. However, all interpolants of the tree interpolant must use the same order.

Theorem 2. *If I_A^1 is a partial tree interpolant of $C_1 \vee \ell$ and I_A^2 is a partial tree interpolant of $C_2 \vee \neg\ell$, then I_A^3 as computed above, after the removal of unsupported variables, is a partial tree interpolant of $C_1 \vee C_2$.*

The proof for this theorem is given in [14].

Example 4 (Resolution). Take the running example and suppose $\ell \equiv g(h(b)) = b$ is the pivot, $I_{\{1\}}^1 \equiv v_{g(h(b))} \leq v_b$ and $I_{\{1\}}^2 \equiv \top$. The interpolants are combined as $I_{\{1\}}^1 \wedge I_{\{1\}}^2$ since $\mathit{colour}(\ell) \notin \{1\}$. This results in the interpolant $v_{g(h(b))} \leq v_b$. After the resolution step, we assume that $v_{g(h(b))}, v_{h(b)}, v_b$ are no longer supported. The outermost variable is $v_{g(h(b))}$, which must be replaced by its definition: $g(v_{h(b)}) \leq v_b$. Now $v_{h(b)}$ is bound by a quantifier, and since h only occurs in partition 1, an existential quantifier is used: $\exists y.\, g(y) \leq v_b$. In the final step, v_b is bound by a universal quantifier since b does not occur in 1, yielding $\forall x.\exists y.\, g(y) \leq x$.

Note that the order of eliminating variables is important. If v_b had been chosen in the first step despite occurring in $FlatEQ(g(h(b)))$, the resulting formula would have been $\exists y.\forall x.g(y) \leq x$. This formula is not logically equivalent and is indeed not a valid interpolant, as it does not follow from $\forall x.g(h(x)) \leq x$.

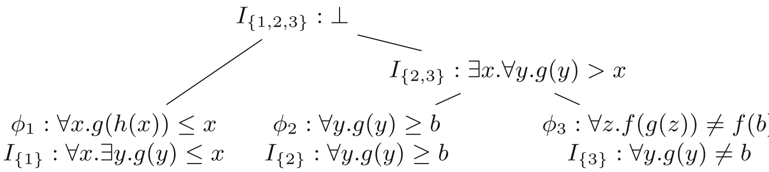


Fig. 1. Tree interpolation problem from Example 1 annotated by tree interpolants.

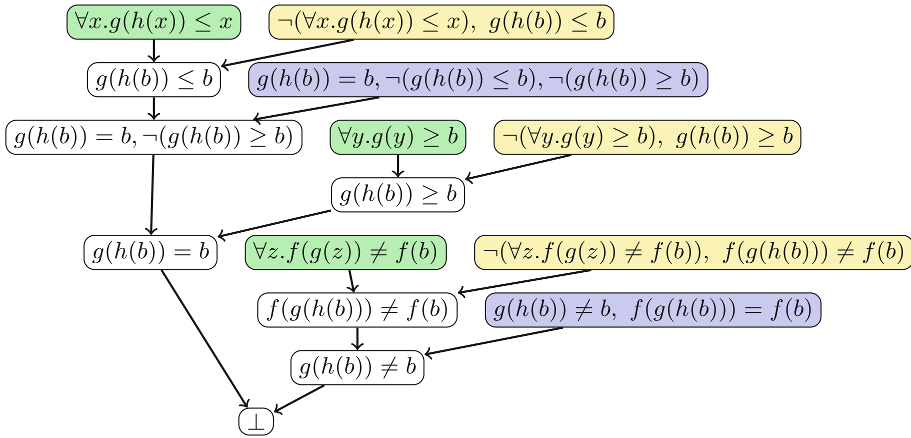


Fig. 2. Resolution proof for Example 1 with input clauses, instantiation lemmas, theory lemmas, and resolvents.

Theorem 3. *The algorithm in this section computes valid tree interpolants from a proof of unsatisfiability.*

Proof. By induction, every node in the proof tree is labelled by a valid partial tree interpolant: Theorem 1 is the base case and Theorem 2 the inductive step. The proof of unsatisfiability ends with the empty clause and its partial interpolant is a tree interpolant for the original problem.

5.2 Full Interpolation Example

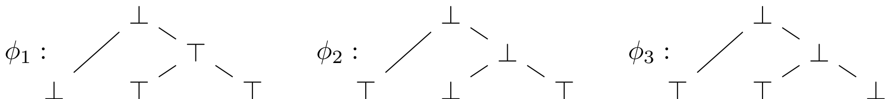
We illustrate the algorithm on our running example (Example 1). Consider the tree interpolation problem given in Fig. 1. The symbol b occurs in partitions 2

and 3, f in 3, g in 1, 2, and 3, and h in 1. Our goal is to compute tree interpolants $I_{\{1\}}$, $I_{\{2\}}$, and $I_{\{3\}}$ for the leaf nodes such that ϕ_1 implies $I_{\{1\}}$, ϕ_2 implies $I_{\{2\}}$, and ϕ_3 implies $I_{\{3\}}$, and tree interpolant $I_{\{2,3\}}$ such that $I_{\{2,3\}}$ is implied by $I_{\{2\}} \wedge I_{\{3\}}$, and $I_{\{1\}} \wedge I_{\{2,3\}}$ implies $\bot$.

Figure 2 shows an instantiation-based resolution proof for the unsatisfiability of $\phi_1 \wedge \phi_2 \wedge \phi_3$. First, we assign each literal occurring in the proof tree to exactly one partition. We colour each proxy literal for a quantified formula by a partition in which it occurs, e.g., $\mathit{colour}(\forall x. g(h(x)) \leq x) = 1$. For the other literals, we can choose arbitrary colours. We assign the literals $g(h(b)) = b$, $g(h(b)) \leq b$, and $g(h(b)) \geq b$ to partition 2, and the literal $f(g(h(b))) \neq f(b)$ to partition 3. We then compute for each literal ℓ the projection onto each partition, i.e., $\ell \downharpoonleft p_i$. For $\ell \equiv g(h(b)) \leq b$ assigned to partition 2, the projections are given in Example 3. As $g(h(b)) \geq b$ and $g(h(b)) = b$ are assigned to the same partition as ℓ and only differ in the comparison operator, their projections only differ in the comparison operator of the flattened version of the original literal. For the remaining literal $f(g(h(b))) = f(b)$, we get the following projections:

$$
\begin{aligned}
f(g(h(b))) = f(b) \downharpoonleft 1 &\equiv v_{g(h(b))} = g(v_{h(b)}) \wedge v_{h(b)} = h(v_b) \\
f(g(h(b))) = f(b) \downharpoonleft 2 &\equiv v_{g(h(b))} = g(v_{h(b)}) \wedge v_b = b \\
f(g(h(b))) = f(b) \downharpoonleft 3 &\equiv v_{f(g(h(b)))} = v_{f(b)} \wedge v_{f(g(h(b)))} = f(v_{g(h(b))}) \wedge \\
&\quad\; v_{g(h(b))} = g(v_{h(b)}) \wedge v_{f(b)} = f(v_b) \wedge v_b = b
\end{aligned}
$$

We now compute partial tree interpolants for each node in the proof tree. The first input clause $C \equiv \phi_1$ on the top left of the proof tree is from partition 1. The partial interpolants $I_{\{1\}}$ and $I_{\{1,2,3\}}$ are set to $\neg(\neg C \downharpoonleft^- A^c) \equiv \bot$, and $I_{\{2\}}$, $I_{\{3\}}$, and $I_{\{2,3\}}$ are set to $\neg C \downharpoonleft^- A \equiv \top$. For the input clauses ϕ_2 and ϕ_3, the interpolants are computed analogously. To summarise:



We now compute the partial tree interpolants for the instantiation lemma on the top right of the proof tree. Similar as for the input clauses, we set $I_{\{1\}}$ to $\neg(\neg C \downharpoonleft^- A^c)$, i.e., to $\neg(\neg C \downharpoonleft^- 2) \wedge \neg(\neg C \downharpoonleft^- 3) \equiv v_{g(h(b))} \leq v_b$. Analogously, we compute all other partial tree interpolants for the three instantiation lemmas:

$\neg\phi_1 \vee g(h(b)) \leq b$:
⊥
$v_{g(h(b))} \leq v_b$ $\quad$ $v_{g(h(b))} > v_b$
$v_{g(h(b))} > v_b$ $\quad$ ⊤

$\neg\phi_2 \vee g(h(b)) \geq b$:
⊥
⊤ ⊥
⊥ ⊤

$\neg\phi_3 \vee f(g(h(b))) \neq f(b)$:
⊥
⊤ ⊥
⊤ ⊥

For the trichotomy lemma, the partial tree interpolants can be set to $\neg C \downharpoonleft^- A$ or $\neg(\neg C \downharpoonleft^- A^c)$. Due to our colouring, all literals in the lemma are either in A

or in A^c. To get the most simple partial interpolants, we set $I_{\{1\}}$ and $I_{\{3\}}$ to $\neg C \downharpoonright^- A \equiv \top$, and $I_{\{2\}}$ and $I_{\{2,3\}}$ to $\neg(\neg C \downharpoonright^- A^c) \equiv \bot$:

$g(h(b)) = b \vee \neg(g(h(b)) \leq b) \vee \neg(g(h(b)) \geq b)$:

$\bot$
$\bot$
$\top$ $\bot$ $\top$

For the congruence lemma, we have $p_f = 3$. The partial tree interpolants $I_{\{1\}}$ and $I_{\{2\}}$ are set to $\neg C \downharpoonright^- A$ as $p_f \notin A$ for these partitions. We get $I_{\{1\}} \equiv \top$ (neither of the flattened literals in $\neg C$ is contained in the projection kernel) and $I_{\{2\}} \equiv v_{g(h(b))} = v_b$, since we chose 2 as the colour of this literal. Similarly, $I_{\{3\}}$ and $I_{\{2,3\}}$ are set to $\neg(\neg C \downharpoonright^- A^c)$. We get $I_{\{3\}} \equiv v_{g(h(b))} \neq v_b$ and $I_{\{2,3\}} \equiv \bot$:

$g(h(b)) \neq b \vee f(g(h(b))) = f(b)$:

$\bot$
$\bot$
$\top$ $v_{g(h(b))} = v_b$ $v_{g(h(b))} \neq v_b$

Having computed the partial tree interpolants for all leaves in the proof tree, we now compute the partial tree interpolants for each resolvent. If the colour of the pivot literal ℓ is in the A-part, i.e., $colour(\ell) \in A$, the partial tree interpolant of the resolvent is the disjunction of the partial tree interpolants of its antecedents. Otherwise, if $colour(\ell) \in A^c$, we build the conjunction of the partial tree interpolants of its antecedents. In the resolution step for the resolvent clause $C_3 \equiv g(h(b)) \leq b$, the pivot literal is assigned to partition 1, i.e., $colour(\forall x.g(h(x)) \leq x) = 1$. To obtain $I_{\{1\}}$, we hence build the disjunction of the partial interpolants of the antecedents $C_1 \equiv \forall x.g(h(x)) \leq x$ and $C_2 \equiv \neg(\forall x.g(h(x)) \leq x) \vee\ g(h(b)) \leq b$, so we get $I_{\{1\}} \equiv I^1_{\{1\}} \vee I^2_{\{1\}} \equiv v_{g(h(b))} \leq v_b$. Similarly, we obtain $I_{\{2\}}$, $I_{\{3\}}$ and $I_{\{2,3\}}$ by conjoining the respective partial interpolants. Since the top-left interpolant is only $\top$ or $\bot$ and the colouring of the pivot literal ensures that we either build the conjunction with $\top$ or the disjunction with $\bot$, the resulting tree interpolant of the resolvent is the same as for the top-right clause. The variables $v_{g(h(b))}$ and v_b are both supported by C_3 and thus allowed to appear in the partial interpolant. The resolution steps of the other inner nodes are very similar in that their partial interpolants always equal the partial interpolant of one of their antecedents. To summarise:

$g(h(b)) \leq b$:
$g(h(b)) = b \vee \neg(g(h(b)) \geq b)$:
$g(h(b)) = b$:

$\bot$
$v_{g(h(b))} > v_b$
$v_{g(h(b))} \leq v_b$ $v_{g(h(b))} > v_b$ $\top$

$g(h(b)) \geq b$:

$\bot$
$\bot$
$\top$ $\bot$ $\top$

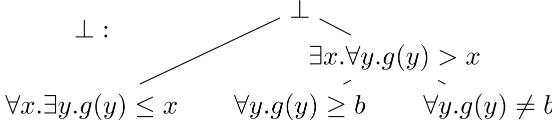
$f(g(h(b))) \neq f(b)$:

$\bot$
$\bot$
$\top$ $\top$ $\bot$

$g(h(b)) \neq b$:

$\bot$
$\bot$
$\top$ $v_{g(h(b))} = v_b$ $v_{g(h(b))} \neq v_b$

The last resolution step is a bit more involved. We have already computed the tree interpolant for partition 1 in Example 4 as $I_{\{1\}} \equiv \forall x.\exists y.g(y) \leq x$. For partition 2, the disjunction $v_{g(h(b))} > v_b \vee v_{g(h(b))} = v_b$ can be simplified to $v_{g(h(b))} \geq v_b$. The outermost variable $v_{g(h(b))}$ is then replaced by $g(v_{h(b)})$, since g occurs in 1 and 2. Then for $v_{h(b)}$ a universal quantifier is introduced, since h only occurs in partition 1, resulting in $\forall y.g(y) \geq v_b$. Finally, v_b is replaced by b, since it occurs in both 2 and 3. This results in $I_{\{2\}} \equiv \forall y.g(y) \geq b$. We omit the computation of the partial interpolant for partitions 3 and the node 23. The partial tree interpolant computed in this step is the tree interpolant of the full interpolation problem:



6 Combination with Equality-Interpolating Theories

In Sects. 4 and 5, we assign each literal to exactly one partition, such that we can apply McMillan's algorithm to combine partial interpolants of the antecedents to obtain a partial interpolant for the resolvent. In the presence of equality-interpolating theories [25], we can also allow for *mixed* literals where only outermost terms must be assigned to one partition. More precisely, we can allow for equalities $t_1 = t_2$ where the left-hand side t_1 is in one partition and the right-hand side t_2 in another, or linear constraints of the form $c_1 \cdot t_1 + \ldots + c_n \cdot t_n \diamond c_0$ with constants c_i and $\diamond \in \{=, \leq, <, \geq, >\}$, where each t_i is assigned to one partition. Such literals can be treated by applying proof tree preserving tree interpolation [5].

A mixed literal $\ell \equiv t_1 = t_2$ is coloured with two colours p_1 and p_2, so that each colour can be chosen to contain the outermost symbols of t_1 and t_2, respectively. The projections are $\ell \downharpoonleft^- p_1 \equiv v_{t_1} = v_\ell$, $\ell \downharpoonleft^- p_2 \equiv v_\ell = v_{t_2}$ and for the negated literal $\neg\ell \downharpoonleft^- p_1 \equiv EQ_1(v_\ell, v_{t_1})$ and $\neg\ell \downharpoonleft^- p_2 \equiv EQ_2(v_\ell, v_{t_2})$, where v_ℓ is a fresh variable and EQ_1, EQ_2 are shared uninterpreted predicates with $\forall x, y.\neg(EQ_1(x, y) \wedge EQ_2(x, y))$, that are only used for the interpolation algorithm. The partial interpolants for a lemma containing mixed literals will contain the auxiliary variable v_ℓ. If a negated mixed equality occurs in the conflict (the negated lemma), we further require that v_ℓ occurs only in literals of the form $EQ_i(v_\ell, s)$ for some shared term s. Valid interpolants will naturally have this shape, as the interpolated conflict also contains v_ℓ only as first parameter of an EQ_i. We then introduce a new combination rule in the first part of interpolating resolution steps: For a mixed literal ℓ, the two interpolants $I^1[EQ_i(v_\ell, s)]$ and $I^2(v_\ell)$ are combined to $I^1[I^2(s)]$, i.e., interpolant $I^2(s)$ replaces the EQ-literals occurring in the interpolant I^1 to form the resolvent interpolant. This eliminates the variable v_ℓ without introducing a quantifier. The remaining part is unchanged, i.e., we still introduce quantifiers for unsupported flattening variables. A proof that the first step produces a valid resolvent interpolant can be

found in [5]. This method produces quantifier-free interpolants if the input formulas were quantifier-free. An example for this method can be found in [13].

7 Implementation in SMTInterpol

We implemented the algorithm in SMTInterpol[2] [6] with a few alterations. First, we used the combination with equality-interpolating theories described in the previous section. Second, we do not apply flattening explicitly. Instead of using an auxiliary variable, the interpolation algorithms for the lemmas include the corresponding term directly. This may result in an interpolant where the interpolant has symbols that are not allowed, because the auxiliary variable was shared but its corresponding function symbol is local to one partition. Only in that case, we introduce the fresh variables for these subterms and replace the offending subterm in the interpolant with its variable. This creates the same interpolants as our presented algorithm, because the latter replaces each variable that stands for a shared function symbol by its definition in the end.

SMTInterpol also supports literals that are shared. If this is done naïvely, the computed interpolants may violate the tree inductivity property (third property in Definition 1). We solve this by treating each literal as occurring in one designated partition when interpolating a lemma (minimizing the number of alternating chains in transitivity lemmas). We then apply Pudlák's resolution rule [21] that has a case for shared literals. Our implementation colours input literals with all partitions it occurs in. For new terms created in the proof, the colour that matches the most outermost function symbols is chosen. If the term uses only symbols from one partition, then it is coloured with that partition. Equalities and inequalities between terms of different partitions are handled with the equality-interpolating procedure to avoid introducing quantifiers when it is not necessary.

8 Conclusion

We presented a tree interpolation algorithm for SMT formulas with quantifiers. The key idea is to virtually flatten each conflict corresponding to a clause in the resolution proof, such that the literals in the flattened version are non-mixed and can be assigned to the different partitions. The colouring of the original literals can even be chosen arbitrarily. Depending on the assigned colours, partial interpolants may contain flattening variables that bridge different partitions, which eventually must be bound by quantifiers.

Our algorithm computes tree interpolants from a single, non-local proof of unsatisfiability obtained independently of the partitioning of the interpolation problem. It supports quantifiers and arbitrary SMT theories, given that the

[2] Official webpage: https://ultimate.informatik.uni-freiburg.de/smtinterpol/
Code available under LGPLv3 at https://github.com/ultimate-pa/smtinterpol.

theory itself supports tree interpolation for its lemmas, and we provided the algorithms for the theory of equality and the theory of linear rational arithmetic.

Correctness proofs for our algorithm are available in [14]. The algorithm is implemented in the open-source SMT solver SMTInterpol.

References

1. Blanc, R., Gupta, A., Kovács, L., Kragl, B.: Tree interpolation in vampire. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR 2013. LNCS, vol. 8312, pp. 173–181. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45221-5_13
2. Bonacina, M.P., Johansson, M.: On interpolation in automated theorem proving. J. Autom. Reason. **54**(1), 69–97 (2015)
3. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2005). https://doi.org/10.1007/11609773_28
4. Christ, J., Hoenicke, J.: Instantiation-based interpolation for quantified formulae. In: Decision Procedures in Software, Hardware and Bioware. Dagstuhl Seminar Proceedings, vol. 10161. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany (2010)
5. Christ, J., Hoenicke, J.: Proof tree preserving tree interpolation. J. Autom. Reasoning **57**(1), 67–95 (2016)
6. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: an interpolating SMT solver. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 248–254. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31759-0_19
7. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. Symb. Log. **22**(3), 269–285 (1957)
8. Dillig, I., Dillig, T., Aiken, A.: Cuts from proofs: a complete and practical technique for solving linear inequalities over integers. Formal Methods Syst. Des. **39**(3), 246–260 (2011)
9. Gleiss, B., Kovács, L., Suda, M.: Splitting proofs for interpolation. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 291–309. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_18
10. Gupta, A., Popeea, C., Rybalchenko, A.: Predicate abstraction and refinement for verifying multi-threaded programs. In: POPL, pp. 331–344. ACM (2011)
11. Gurfinkel, A., Rollini, S.F., Sharygina, N.: Interpolation properties and SAT-based model checking. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 255–271. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02444-8_19
12. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: POPL, pp. 471–482. ACM (2010)
13. Henkel, E., Hoenicke, J., Schindler, T.: Proof tree preserving sequence interpolation of quantified formulas in the theory of equality. In: SMT. CEUR Workshop Proceedings, vol. 2908, pp. 3–16. CEUR-WS.org (2021)
14. Henkel, E., Hoenicke, J., Schindler, T.: Choose your colour: tree interpolation for quantified formulas in SMT. CoRR abs/2305.11667 (2023). https://arxiv.org/abs/2305.11667
15. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL, pp. 232–244. ACM (2004)

16. Hoenicke, J., Majumdar, R., Podelski, A.: Thread modularity at many levels: a pearl in compositional verification. In: POPL, pp. 473–485. ACM (2017)
17. Kovács, L., Voronkov, A.: First-order interpolation and interpolating proof systems. In: LPAR. EPiC Series in Computing, vol. 46, pp. 49–64. EasyChair (2017)
18. McMillan, K.L.: An interpolating theorem prover. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 16–30. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_2
19. McMillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 413–427. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_31
20. Monniaux, D., Gonnord, L.: Cell morphing: from array programs to array-free horn clauses. In: Rival, X. (ed.) SAS 2016. LNCS, vol. 9837, pp. 361–382. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_18
21. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. J. Symb. Log. **62**(3), 981–998 (1997)
22. Rollini, S.F., Bruttomesso, R., Sharygina, N.: An efficient and flexible approach to resolution proof reduction. In: Barner, S., Harris, I., Kroening, D., Raz, O. (eds.) HVC 2010. LNCS, vol. 6504, pp. 182–196. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19583-9_17
23. Rümmer, P., Hojjat, H., Kuncak, V.: On recursion-free horn clauses and Craig interpolation. Formal Methods Syst. Des. **47**(1), 1–25 (2015)
24. Seghir, M.N., Podelski, A., Wies, T.: Abstraction refinement for quantified array assertions. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 3–18. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03237-0_3
25. Yorsh, G., Musuvathi, M.: A combination method for generating interpolants. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 353–368. Springer, Heidelberg (2005). https://doi.org/10.1007/11532231_26

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.