Instantiation-Based Interpolation for Quantified Formulae

Jürgen Christ
Albert-Ludwigs-Universität,
79110 Freiburg, Germany
christj@informatik.uni-freiburg.de

Jochen Hoenicke
Albert-Ludwigs-Universität,
79110 Freiburg, Germany
hoenicke@informatik.uni-freiburg.de

Abstract

Interpolation has proven highly effective in program analysis and verification, e. g., to derive invariants or new abstractions. While interpolation for quantifier free formulae is understood quite well, it turns out to be challenging in the presence of quantifiers. We present in this paper modifications to instantiation based SMT-solvers and to McMillan's interpolation algorithm in order to compute quantified interpolants.

1 Introduction

While SMT solvers have been used in verification and model checking for many years [11], recent approaches use Craig interpolation [3], e. g., to compute new predicates during the refinement step of predicate abstraction [18] or to compute image operators in symbolic model checking [22, 19, 24, 26, 7, 16, 17]. Model checkers like ARMC [26] or SLAB [7] for example use interpolation during their abstraction refinement phase. Furthermore, interpolants can be used in automated invariant generation [25].

Current interpolation algorithms either are not able to handle quantified input formulae [23, 27] or cannot be integrated into recent SMT solvers based on quantifier instantiation [3, 25, 20]. As verification requires the ability to reason about quantifiers which arise from various sources like framing axioms or theories that are not equipped with an own decision procedure, interpolation procedures that can deal with quantified input formulae are needed.

We combine instantiation based proof generators with McMillan's interpolation algorithm presented in [23]. The combination allows arbitrary quantifier alternation in its input and produces a valid Craig interpolant. It can easily be integrated into existing SMT solvers [5, 13, 10] and adapted to different interpolation schemes [9, 8]. To our knowledge, this is the first method that allows for efficient interpolation of quantified formulae from an instantiation based resolution proof given by a state of the art SMT solver.

The contributions of this paper are a generic, satisfiability preserving modification to quantifier instantiation procedures and an additional step during resolution to generate a valid interpolant.

2 Notation

We assume the usual notions and terminology of first order logic. Let Σ be a *signature* consisting of a set of *function* and *predicate* symbols. Each symbol is assigned a non-negative *arity*. We call 0-ary function symbols *constants* and denote them by a, b, and s. In the context of quantifiers we also consider 0-ary *variable symbols* which we will denote by x, y, and z. We abbreviate sets of variables bound by one quantifier by $\bar{x}$.

A *theory* $\mathcal{T}$ is a set of formulae (axioms). For two formulae ϕ and ψ, we say $\phi \models_{\mathcal{T}} \psi$ if and only if $\mathcal{T} \models \phi \rightarrow \psi$. Formula ϕ is *$\mathcal{T}$-unsatisfiable*, if and only if $\phi \models_{\mathcal{T}} \perp$ and a *$\mathcal{T}$-tautology* if and only if its negation is $\mathcal{T}$-unsatisfiable. We suppress the subscript $\mathcal{T}$ whenever the theory is clear from context or the result is independent of a theory.

We call a quantifier in formula ϕ *positive* if it is not negated after converting ϕ into Negation Normal Form (NNF) and *negative* otherwise. Each quantifier binds a positive number of distinct variables.

For a formula ϕ, the set $syms(\phi)$ contains all function and predicate symbols occurring in ϕ. The set $vars(\phi)$ contains all unbound variables occurring in the formula. With a slight abuse of notation, given a term t, we use $syms(t)$ and $vars(t)$ to denote the set of symbols resp. the set of free variables in t.

Definition 1. Given a theory $\mathscr{T}$ and two closed formulae, A and B, whose conjunction is $\mathscr{T}$-unsatisfiable, a *Craig Interpolant* is a closed formula I such that

1. $A \models_{\mathscr{T}} I$
2. $B \wedge I$ is $\mathscr{T}$-unsatisfiable and
3. $syms(I) \subseteq (syms(A) \cap syms(B)) \cup syms(\mathscr{T})$.

Given two formulae A and B in CNF, we decompose every clause C occurring in the resolution refutation of $A \wedge B$ into the set of literals that occur only in A (denoted by $\mathsf{local}_A(C)$), the set of literals occurring only in B (denoted by $\mathsf{local}_B(C)$), and the set $\mathsf{shared}(C)$ of literals that occur in A and B or in the theory $\mathscr{T}$. We use $\mathsf{in}_B(C)$ to abbreviate $\mathsf{shared}(C) \cup \mathsf{local}_B(C)$.

Let $C_1 \vee l$ and $C_2 \vee \neg l$ be two clauses. Applying the resolution rule yields the *resolvent clause* $C_1 \vee C_2$. The literal l is called the *pivot* of the resolution. A *resolution proof* or *resolution refutation* is a sequence of applications of the resolution rule to input clauses and resolvents that results in the empty clause.

3 Preliminaries

This section presents basic techniques which we extend and combine in section 4. In 3.1, we present a strategy to prove unsatisfiability of quantified formulae. Section 3.2 describes McMillan's interpolation algorithm [23].

3.1 Instantiation Based Refutation

This section gives an overview of state of the art SMT solving. Since we cannot compute interpolants for satisfiable formulae, we bias our presentation towards refutations.

To prove unsatisfiability of a formula ϕ, we first transform ϕ into conjunctive normal form (CNF) using a Tseitin-style encoding [29]. We use skolemization for every positive existential and for every negative universal quantifier. Remaining quantified formulae get transformed into *proxy literals* [6]. Those proxy literals are propositional abstractions of the corresponding quantified formulae. I. e., for a formula $\forall x.\ \phi(x)$, we introduce a proxy $l_{\forall x.\phi(x)}$, which represents the truth value of $\forall x.\ \phi(x)$.

Next, we use DPLL($\mathscr{T}$) [12] as proof search strategy to check the skolemized formula for satisfiability. The set of clauses used as input to DPLL($\mathscr{T}$) contains only ground and proxy literals. Quantifier reasoning is done by an instantiation strategy which, given a quantified formula, detects instantiations needed to prove unsatisfiability of the input formula. Whenever a proxy literal $l_{\forall x.\phi(x)}$ is asserted, an instantiation procedure is used to produce instantiations for the formula $\forall x.\phi(x)$. For every instantiation, an instantiation clause is added which connects the proxy literal representing the quantified formula with the instantiation. If, e. g., the proxy literal $l_{\forall x.\phi(x)}$ is asserted and the formula $\forall x.\phi(x)$ gets instantiated by $x \leftarrow c$, the clause $\neg l_{\forall x.\phi(x)} \vee \phi(c)$ is added to the proof system.

Instantiation procedures might, e. g., be guided by patterns [6] or by instantiation sets based on a structural analysis of the input formula like array properties [1] or local theory extensions [14, 21]. All these methods replace a quantified formula $\forall \bar{x}.\phi(\bar{x})$ by a finite conjunction $\bigwedge_{\bar{c}_i} \phi(\bar{c}_i)$.

In the remainder of this paper, we do not assume any specific instantiation procedure. Every procedure that replaces a quantified formula by a conjunction of instantiations can be modified as described in 4.1.

Example 1 (Refutation of a quantified formula)**.** Take the formula

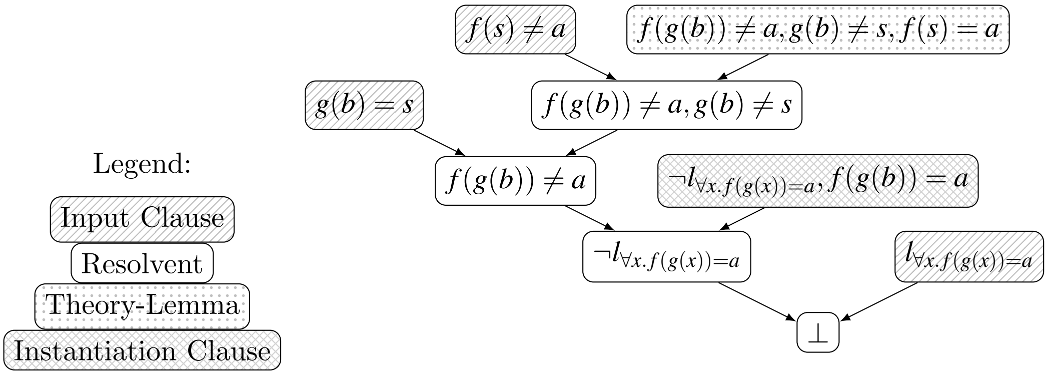
$$(\forall x. f(g(x)) = a) \wedge f(s) \neq a \wedge g(b) = s.$$

It first gets transformed into the clause set

$$\{\{l_{\forall x. f(g(x))=a}\}, \{f(s) \neq a\}, \{g(b) = s\}\}$$

which only contains unit clauses. Hence, every literal gets asserted and, since $l_{\forall x. f(g(x))=a}$ is asserted, an instantiation procedure is used to produce instantiations for $\forall x. f(g(x)) = a$. This procedure then detects the instantiation $x \leftarrow b$ which is needed to prove unsatisfiability of the original formula.

If we instantiate $x \leftarrow b$, we get a resolution refutation like presented in Figure 1. Dotted shading indicates additional theory lemmata that were derived during proof generation, north east shading depicts input clauses, and crosshatch shaded boxes represent instantiation clauses.



$f(s) \neq a$
$f(g(b)) \neq a, g(b) \neq s, f(s) = a$
$g(b) = s$
$f(g(b)) \neq a, g(b) \neq s$
Legend:
$f(g(b)) \neq a$
$\neg l_{\forall x. f(g(x))=a}, f(g(b)) = a$
Input Clause
Resolvent
Theory-Lemma
Instantiation Clause
$\neg l_{\forall x. f(g(x))=a}$
$l_{\forall x. f(g(x))=a}$
$\perp$

Figure 1: Refutation by resolution for example 1

3.2 Quantifier Free Interpolation

Algorithms to compute interpolants from resolution proofs of unsatisfiability were proposed, e. g., in [27] or [23]. These algorithms both assign a partial interpolant to every clause occurring in the proof. Correctness of these algorithms is ensured by an invariant for partial interpolants. We will here give a short overview over McMillan's version presented in [23].

To compute an interpolant from a resolution refutation of $A \wedge B$, the algorithm annotates every clause with a partial interpolant. For input clauses C in A, its partial interpolant is set to $\mathsf{shared}(C)$. Clauses in B are annotated with $\top$. Theory specific interpolators annotate theory lemmata with partial interpolants. Every resolution step combines the partial interpolants of the premises depending on the pivot element: If the pivot occurs in at least one clause in B, the partial interpolants from the input clauses are combined by conjunction, otherwise by disjunction. The partial interpolant assigned to the empty clause is returned as Craig Interpolant.

Theorem 1 (Correctness of McMillan's Interpolation Algorithm)**.** *McMillan's interpolation algorithm constructs valid Craig interpolants.*

A proof for Theorem 1 is given in [23]. This proof relies on following invariants for a clause C with partial interpolant I_C:

$$\begin{array}{rcl} A & \models_{\mathscr{T}} & I_C \vee \mathsf{local}_A(C) \\ B \wedge I_C & \models_{\mathscr{T}} & \mathsf{in}_B(C) \\ syms(I_C) & \subseteq & (syms(A) \cap syms(B)) \cup syms(\mathscr{T}) \end{array}$$

4 Interpolation for Quantified Formulae

This section presents the contributions of this paper. It combines quantifier instantiation with McMillan's interpolation procedure to derive quantified interpolants for any formula that can be proven unsatisfiable. First, in 4.1, we show how to modify instantiations in order to support interpolation. Next, in 4.2 we describe a method to infer quantified interpolants whenever they are needed. Finally, in 4.3, we give a correctness and termination proof of our algorithm.

4.1 Instance Purification

An interpolation problem consists of two closed formulae A and B. Interpolants as defined in definition 1 only contain symbols shared between A and B. Every term occurring in an input clause either is local to A if it contains at least one symbol that does not occur in B, local to B if it contains at least one symbol that does not occur in A, or shared. Instantiation produces new terms by substituting variables. Resulting terms might neither be local to A or B, nor shared. If the variable, e. g., is an argument to an A-local function symbol and it gets substituted by a B-local term, the resulting term contains a mixture of local symbols and is neither local to A or B, nor shared. We call such a term mixed.

For resolution based interpolation algorithms, it is important to have a function symbol and all its arguments in the same partition. Otherwise, we get a mixed term which is difficult to use during interpolation. But we do not want to restrict an instantiation procedure to only instantiate quantifiers such that the instantiation does not create a mixed term. This would significantly reduce the number of formulae we could refute. Hence we give an algorithm to purify an instantiation term according to the partition of the quantifier which gets instantiated. This step introduces new variables which are shared between both parts of an interpolation problem and auxiliary equalities assigning meaning to these variables.

Inputs to purification are an instantiation term $t \equiv f(t_1,\ldots,t_n)$ (recall that constants are 0-ary functions) and a target partition $P \in \{\mathrm{A},\mathrm{B}\}$. The algorithm proceeds as follows:

1. If f occurs in the target partition, return f applied to the instance purification of $t_1,\ldots,t_n$ onto P.
2. Otherwise, f occurs in the other partition P'. Create a fresh variable that represents f. Add to P' the equality between this variable and the application of f to $t_1,\ldots,t_n$ purified onto P'. Return the newly created variable.

Example 2 (Instance Purification)**.** Let $f/1$ and a be symbols occurring only in partition A and $g/2$ and b symbols occurring only in B. Tables below show the purification of $f(g(a,b))$ onto A resp. B. They show every step in the algorithm including the term t currently purified, the target partition P, the additional equalities (Aux EQs) introduced during this step, and the result.

t	P	Aux EQs	Result
$f(g(a,b))$	A		$f(v_2)$
$g(a,b)$	A	$v_2 = g(v_1,b)$	v_2
a	B	$v_1 = a$	v_1
b	B		b

t	P	Aux EQs	Result
$f(g(a,b))$	B	$v_3 = f(v_2)$	v_3
$g(a,b)$	A	$v_2 = g(v_1,b)$	v_2
a	B	$v_1 = a$	v_1
b	B		b

4.2 Quantifier Introduction

The instance purification described above introduces new variables which are shared between both partitions and hence may occur in an interpolant. Since these variables are not part of the original input formula, we have to remove them by introducing quantifiers. Although it would be possible to create an unquantified interpolant containing auxiliary variables and introduce the quantifiers as last step [23] (yielding Prenex Normal Form), we propose a modification to the interpolation engine in order to introduce quantifiers as soon as possible. This has some impact on heuristic instantiation procedures: After skolemization of a formula in Prenex Normal Form, we end up with one universal quantifier binding some variables. Our decision to introduce quantifiers early can prevent this setting and, therefore, generates an easier problem for instantiation heuristics like ,e. g., E-Matching[6].

We say a literal *supports* a variable, if and only if the variable occurs inside the literal. A variable v_1 depends on v_2 if v_2 occurs in the auxiliary equality defining v_1. If v is defined by the equality $v = t$, the set of variables v depends on is $deps(v) = vars(t)$. Its inverse, $ideps(v)$, denotes all variables v' such that $v \in deps(v')$. By $ideps^*$ we denote the reflexive transitive closure of $ideps$.

We have to order quantifier introduction such that dependencies between variables are translated into dependencies between the quantifiers. If a variable v_1 depends on v_2, an auxiliary equality $v_1 = \phi(v_2)$ exists. This equality tells us to first choose a value for v_2 and then compute the value for v_1. According to this observation, the quantifiers should occur in dependency order. But our interpolation procedure puts quantifiers in front of a previously constructed partial interpolant. Hence, the procedure introduces quantifiers in inverse dependency order which is a topological order on the inverse dependency graph.

One has to be careful if a variable becomes unsupported and at least one variable that depends on it is still supported by the clause. In this case we cannot introduce a quantifier for the unsupported variable immediately, but have to delay introduction until quantifiers for all inverse dependencies of this variable are introduced.

The new algorithm to annotate a resolution tree containing auxiliary variables is shown below. It adds steps 3, 4 and 5c to McMillan's algorithm:

1. Annotate each input clause C with a partial clause interpolant I_C like in McMillan's algorithm.

2. For each theory lemma C_l, generate a theory specific interpolant I_{C_l} (using a theory specific interpolator).

3. For each unit clause U containing an auxiliary equality, set I_U to $\top$ if the equality was added to B and to $\bot$ otherwise.

4. Treat every instantiation clause like a regular input clause.

5. On Resolution of $C_1 \vee l$ with interpolant I_1 and $C_2 \vee \neg l$ with interpolant I_2 to $C_3 \equiv C_1 \vee C_2$ with interpolant I_3:

    (a) If l occurs in B, set I_3 to $I_1 \wedge I_2$. Otherwise, set I_3 to $I_1 \vee I_2$.

    (b) Build quantification set $QS = ideps^*(vars(l)) \setminus ideps^*(vars(C_3))$.

    (c) For each variable $v \in QS$ in inverse dependency order:

        i. If v is mapped to an A-local term, set $I_3 \equiv \exists v.I_3$ and to $I_3 \equiv \forall v.I_3$ otherwise. The quantifier might be omitted if v does not occur in I_3.

Example 3. We use our algorithm to interpolate the formula presented in example 1. We partition the interpolation problem into

$$\begin{aligned} A &\equiv (\forall x. f(g(x)) = a) \land f(s) \neq a \\ B &\equiv g(b) = s \end{aligned}$$

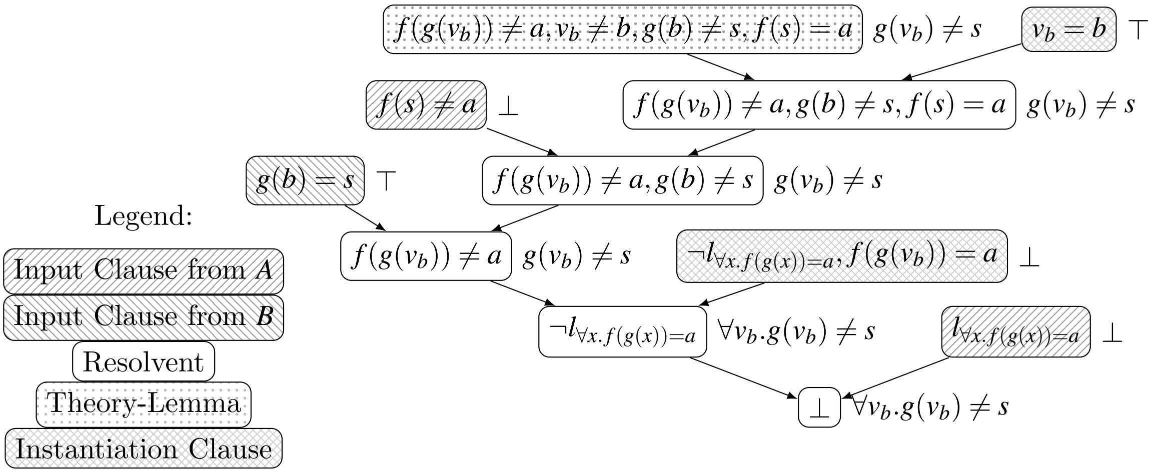

$f(g(v_b)) \neq a, v_b \neq b, g(b) \neq s, f(s) = a$ $g(v_b) \neq s$
$v_b = b$ $\top$
$f(s) \neq a$ $\bot$
$f(g(v_b)) \neq a, g(b) \neq s, f(s) = a$ $g(v_b) \neq s$
$g(b) = s$ $\top$
$f(g(v_b)) \neq a, g(b) \neq s$ $g(v_b) \neq s$
Legend:
Input Clause from A
Input Clause from B
Resolvent
Theory-Lemma
Instantiation Clause
$f(g(v_b)) \neq a$ $g(v_b) \neq s$
$\neg l_{\forall x. f(g(x))=a}, f(g(v_b)) = a$ $\bot$
$\neg l_{\forall x. f(g(x))=a}$ $\forall v_b. g(v_b) \neq s$
$l_{\forall x. f(g(x))=a}$ $\bot$
$\bot$ $\forall v_b. g(v_b) \neq s$

Figure 2: Refutation by resolution for Example 1 annotated by partial interpolants.

Since the quantified formula is in A and the instantiation term b is in B, we introduce a variable v_b and the equality $v_b = b$ to B. The annotated resolution tree is shown in Figure 2. The leaves are shaded according to where the clause originated. North east shading depicts clauses coming from A, north west shaded clauses come from B, theory lemmata are dotted, and crosshatching represents instantiation clauses and auxiliary equalities. It is easy to see that the annotation for the bottom clause is a valid interpolant.

The previous example presents the simple story. It does not include variable dependencies and hence does not have to deal with ordering of quantifier introductions. Hence, we present an example which is more involved and shows the importance of dependency tracking:

Example 4. In this example we will derive an interpolant for

$$\begin{aligned} A &= \forall x.\, h(x) \neq a \\ B &= \forall y.\, h(g(y)) = y \end{aligned}$$

For a refutation, we need to instantiate x with $g(a)$ and y with a. But g is B-local and a is A-local. Hence, our instantiation procedure instantiates x with v_g and y with v_a. Additionally, the equality $v_a = a$ is added to A and $v_g = g(v_a)$ is added to B. Figure 3 shows the resolution tree for one possible refutation. Again, north east shading depicts clauses coming from A, north west shaded clauses come from B, theory lemmata are dotted, and crosshatched ones represent instantiation clauses and auxiliary equalities.

Since v_a has an inverse dependency on v_g, we cannot introduce a quantifier for v_a before a quantifier for v_g is introduced. Note that there is a proof tree which respects inverse dependency ordering and, therefore, does not prevent quantifier introduction (see Figure 4).

During the last resolution step, in figure 3, v_g becomes unsupported and we can add the quantifiers for both v_g and v_a. Inverse order requires to first quantify over v_g (producing $\forall v_g.\, h(v_g) \neq v_a$) and then quantify over v_a. Correctness of the resulting interpolant $\exists v_a.\, \forall v_g.\, h(v_g) \neq v_a$ can easily be verified.

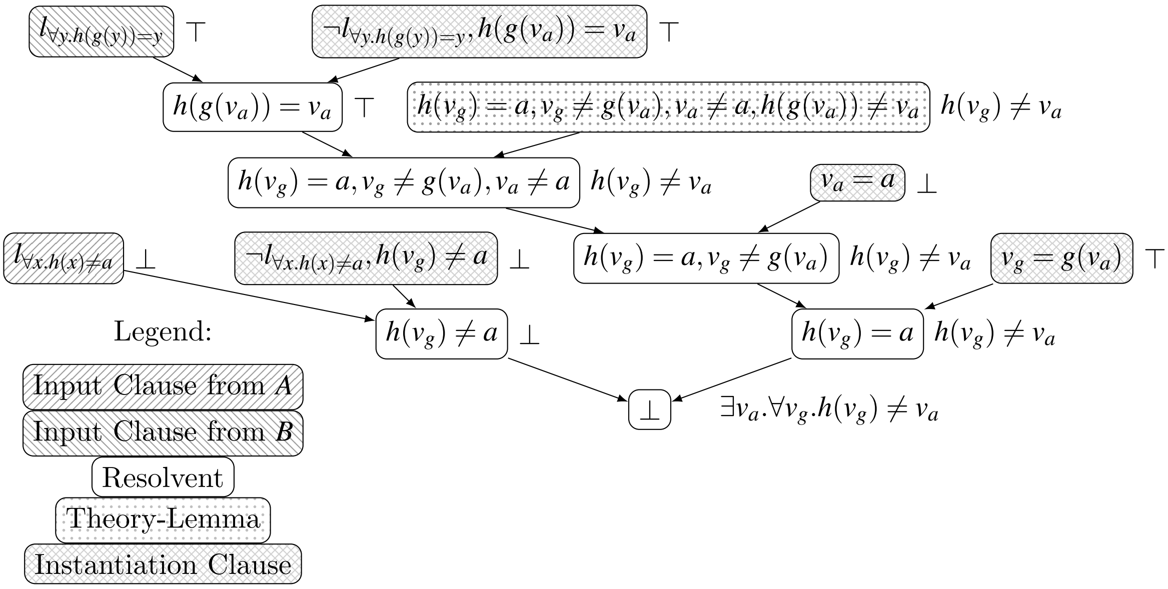

$l_{\forall y.h(g(y))=y}$ $\top$
$\neg l_{\forall y.h(g(y))=y}, h(g(v_a)) = v_a$ $\top$
$h(g(v_a)) = v_a$ $\top$
$h(v_g) = a, v_g \neq g(v_a), v_a \neq a, h(g(v_a)) \neq v_a$ $h(v_g) \neq v_a$
$h(v_g) = a, v_g \neq g(v_a), v_a \neq a$ $h(v_g) \neq v_a$
$v_a = a$ $\bot$
$l_{\forall x.h(x)\neq a}$ $\bot$
$\neg l_{\forall x.h(x)\neq a}, h(v_g) \neq a$ $\bot$
$h(v_g) = a, v_g \neq g(v_a)$ $h(v_g) \neq v_a$
$v_g = g(v_a)$ $\top$
Legend:
$h(v_g) \neq a$ $\bot$
$h(v_g) = a$ $h(v_g) \neq v_a$
Input Clause from A
Input Clause from B
$\bot$ $\exists v_a.\forall v_g.h(v_g) \neq v_a$
Resolvent
Theory-Lemma
Instantiation Clause

Figure 3: Refutation by resolution for Example 4 annotated by partial interpolants.

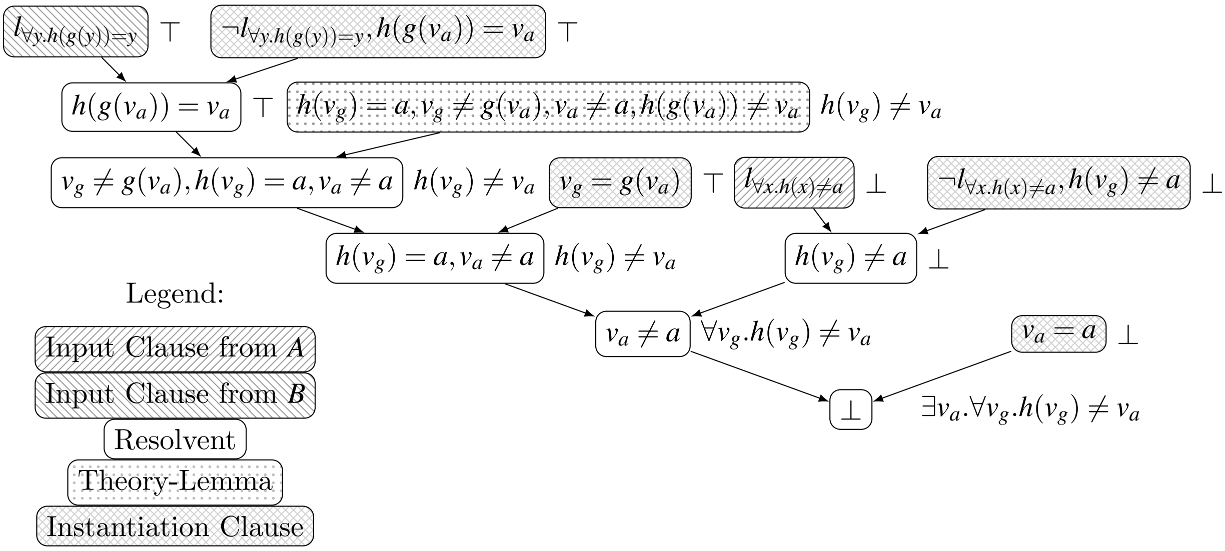

$l_{\forall y.h(g(y))=y}$ $\top$
$\neg l_{\forall y.h(g(y))=y}, h(g(v_a)) = v_a$ $\top$
$h(g(v_a)) = v_a$ $\top$
$h(v_g) = a, v_g \neq g(v_a), v_a \neq a, h(g(v_a)) \neq v_a$ $h(v_g) \neq v_a$
$v_g \neq g(v_a), h(v_g) = a, v_a \neq a$ $h(v_g) \neq v_a$
$v_g = g(v_a)$ $\top$
$l_{\forall x.h(x)\neq a}$ $\bot$
$\neg l_{\forall x.h(x)\neq a}, h(v_g) \neq a$ $\bot$
$h(v_g) = a, v_a \neq a$ $h(v_g) \neq v_a$
$h(v_g) \neq a$ $\bot$
Legend:
$v_a \neq a$ $\forall v_g.h(v_g) \neq v_a$
$v_a = a$ $\bot$
Input Clause from A
Input Clause from B
$\bot$ $\exists v_a.\forall v_g.h(v_g) \neq v_a$
Resolvent
Theory-Lemma
Instantiation Clause

Figure 4: Refutation by resolution for Example 4 annotated by partial interpolants respecting inverse dependency ordering.

Notice that $\forall v_g.\ \exists v_a.\ h(v_g) \neq v_a$ is not a valid interpolant since it is not inconsistent with B. Consider a universe consisting of two distinct elements c_0 and c_1. Furthermore, let h and g be the identify function. We have to show that $\forall y.\ h(g(y)) = y \wedge \forall v_g.\ \exists v_a.\ h(v_g) \neq v_a$ is unsatisfiable. Skolemization yields $\forall y.\ h(g(y)) = y \wedge \forall v_g.\ h(v_g) \neq sk_{v_a}(v_g)$. This formula should be unsatisfiable, but we can extend the interpretation above by $sk_{v_a}(c_i) = c_{1-i}$ and end up with a satisfying model.

4.3 Correctness

In this section, we show correctness of the algorithm presented above. We start by showing that our instantiation modification preserves unsatisfiability and finally show that our algorithm yields a valid interpolant.

Theorem 2 (Purification preserves unsatisfiability). *If a set of quantifier instantiations is sufficient to show unsatisfiability of a formula, the corresponding set of purified instantiations suffices, too.*

Proof for this theorem is trivial since quantifier instantiation and purified quantifier instantiation yield equisatisfiable formulae.

Lemma 1 (Inverse dependency order). Inverse dependencies of variables introduced during purification of instances induce a strict order on these variables.

Proof is trivial since we defined inverse dependency order as topological order on the inverse dependency graph of all auxiliary variables.

Theorem 3 (Termination). *The interpolation procedure presented in this paper terminates if a finite resolution proof is used.*

Proof of this theorem follows from the fact that the resolution proof is finite and that we only add a finite number of quantifiers.

Theorem 4 (Valid Interpolant). *Our method presented above yields a valid interpolant for an unsatisfiable pair of closed formulae (A,B).*

Proof. Let $auxeqs_A(V)$ denote the set of auxiliary equalities in A that define a variable in V and similarly $auxeqs_B(V)$. By induction over the resolution proof one can show that every partial interpolant I_C annotating a clause C satisfies following invariants:

$$\begin{array}{rcl} A \wedge auxeqs_A(ideps^*(vars(C))) & \models_{\mathscr{T}} & \mathsf{local}_A(C) \vee I_C \\ B \wedge auxeqs_B(ideps^*(vars(C))) \wedge I_C & \models_{\mathscr{T}} & \mathsf{in}_B(C) \\ syms(I_C) & \subseteq & (syms(A) \cap syms(B)) \cup syms(\mathscr{T}) \\ vars(I_C) & \subseteq & ideps^*(vars(C)) \end{array}$$

For $C = \perp$ this entails that $I_\perp$ is a Craig Interpolant of A and B. The full proof for this theorem is given in appendix A. □

5 Implementation

We implemented the interpolation algorithm presented in this paper in an experimental interpolating SMT solver written in Java. Our solver supports the theory of equality and uninterpreted functions in combination with linear arithmetic. By reduction to uninterpreted functions, we also support the theory of arrays. We extract instantiations relevant for conjecture of an input problem from Z3 proof trees [4] and introduce them as purified ground formulae into our solver. This solver computes partial interpolants according to the algorithm above during resolution. As optimization, if the same term is purified multiple times, we reuse the auxiliary variable representing this term.

Upon instantiation, we add an instantiation clause and auxiliary equalities produced during purification. For all literals introduced this way, we remember all auxiliary variables occurring in this literal. We propagate this information to clauses and use it during resolution to compute the introduction set for this step. Hence, we need to keep track of all variables supported by a clause. This produces some memory overhead which is reduced by lazy initialization and aggressive sharing of these sets.

6 Related Work

McMillan presented in [23] a short overview of how to infer quantified interpolants. While his presentation only presented an informal proof, we gave a complete proof for a different interpolation system. His system introduces fresh symbols for every instantiation and adds auxiliary equalities. Our approach also captures mixed instantiations where a function symbol comes from a different partition than some of its arguments. His interpolants are in Prenex Normal Form while we push quantifiers as far as possible into the formula. This has different effects on instantiation based solvers (see [6, page 46]).

In [25], McMillan shows how to use saturation based solvers to generate *universally* quantified interpolants. Since we also want to have existentially quantified interpolants, such an interpolation procedure is not powerful enough. Furthermore, instantiation based approaches to deal with quantifiers yield a better performance than saturation based ones on many software model checking problems.

In [20], Kovács and Voronkov give a more general version of interpolation for saturation based solvers than [25]. Their technique uses a separating term ordering which prevents mixed terms during inference. In contrast to our method, they explicitly prevent such terms while we purify all instantiations such that we get a separated proof.

Sofronie-Stokkermans [28] provides an interpolation procedure for a decidable set of theory extensions. She considers quantifier free formulae over local theory extensions. Such extensions provide axioms for new symbols which can then be replaced using a set of rewriting and instantiation rules. Resulting interpolants are quantifier free modulo theory extensions. Our method does not give such guarantees. Nevertheless the algorithm presented in this paper is able to generate an interpolant for all inputs the method described in [28] can compute an interpolant for.

Purification of literals has also been discussed in papers related to interpolation in the combination of different theories, e. g., in [30, 2, 15]. In [30], Yorsh and Musuvathi show how to combine certain theories without using quantifiers. They define equality-interpolating theories, which provide for a mixed interface equality $a = b$ a shared term t such that $a = t$ and $t = b$ hold. While our method can also be used to compute quantified interpolants in the combination of multiple theories, we strongly recommend a combined approach. For a mixed equality that follows from a theory lemma, the shared term should be computed as in [30]. However, for a mixed equality that is created by a quantifier instantiation, a shared term may not always exist, and our method should be used.

7 Conclusions

We presented a method to obtain interpolants for quantified formulae from state of the art SMT solvers based on quantifier instantiation. This method always delivers a valid interpolant if a proof of unsatisfiability can be generated. The main design goal of the method presented in this paper is easy integration into existing solvers based on quantifier instantiation.

Although our presentation focuses on McMillan's interpolation algorithm [23], it is not limited to this approach. Instantiation purification can be done in general and the quantifier introduction step can be done after every interpolation step of resolution based interpolation systems like the generic one presented in [9] which can be instantiated to produce McMillan's or Pudlák's system.

Acknowledgments

This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR14 AVACS). See www.avacs.org for more information.

This work was partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft XT project under grant 01 QS 07 008. The responsibility for this article lies with the authors.

We thank Viorica Sofronie-Stokkermans for helpful comments and discussions.

References

[1] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What 's decidable about arrays? In *VMCAI'06*, volume 3855 of *lncs*, pages 427–442, Charleston, SC, January 2006. Springer Verlag.

[2] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Efficient interpolant generation in satisfiability modulo theories. In *TACAS*, pages 397–412, 2008.

[3] William Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *J. Symb. Log.*, 22(3):269–285, 1957.

[4] Leonardo de Moura and Nikolaj Bjørner. Proofs and refutations, and z3. In *LPAR Workshops*, 2008.

[5] Leonardo M. de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS'08*, pages 337–340, 2008.

[6] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[7] Klaus Dräger, Andrey Kupriyanov, Bernd Finkbeiner, and Heike Wehrheim. SLAB: A certifying model checker for infinite-state concurrent systems. In *TACAS'10*, Lecture Notes in Computer Science. Springer-Verlag, 2010.

[8] Vijay D'Silva. Propositional interpolation and abstract interpretation. In *Proceedings of the European Symposium on Programming (ESOP'10)*. Springer, 2010.

[9] Vijay D'Silva, Daniel Kroening, Mitra Purandare, and Georg Weissenbacher. Interpolant strength. In *VMCAI'10*, pages 129–145, 2010.

[10] Bruno Dutertre and Leonardo De Moura. The YICES SMT solver. Technical report, 2006.

[11] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI'02*, pages 234–245, 2002.

[12] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In *CAV'04*, pages 175–188, 2004.

[13] Yeting Ge, Clark Barrett, and Cesare Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In *CADE'07*, pages 167–182, 2007.

[14] Robert Givan and David A. McAllester. New results on local inference relations. In *KR*, pages 403–412, 1992.

[15] Amit Goel, Sava Krstic, and Cesare Tinelli. Ground interpolation for combined theories. In *CADE*, pages 183–198, 2009.

[16] M. Heizmann, J. Hoenicke, and A. Podelski. Refinement of trace abstraction. In *SAS'09*, number 5673 in LNCS, pages 69–85. Springer, 2009.

[17] M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *POPL'10*, pages 471–482. ACM, 2010.

[18] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *POPL'04*, pages 232–244, 2004.

[19] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *POPL'04*, pages 232–244, 2004.

[20] Laura Kovács and Andrei Voronkov. Interpolation and symbol elimination. In *CADE'09*, pages 199–213, 2009.

[21] David A. McAllester. Automatic recognition of tractability in inference relations. *J. ACM*, 40(2):284–303, 1993.

[22] Kenneth L. McMillan. Interpolation and SAT-based model checking. In *CAV'03*, pages 1–13, 2003.

[23] Kenneth L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.

[24] Kenneth L. McMillan. Lazy abstraction with interpolants. In *CAV'06*, pages 123–136, 2006.

[25] Kenneth L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *TACAS'08*, pages 413–427, 2008.

[26] Andreas Podelski and Andrey Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. In *PADL'07*, pages 245–259, 2007.

[27] Pavel Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations, 1996.

[28] Viorica Sofronie-Stokkermans. Interpolation in local theory extensions, October 2008. Special issue of LMCS dedicated to IJCAR 2006.

[29] G. S. Tseitin. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, pages 466–483. Springer, Berlin, Heidelberg, 1983.

[30] Greta Yorsh and Madanlal Musuvathi. A combination method for generating interpolants. In *CADE*, pages 353–368, 2005.

A Proof of Theorem 4

The proof follows the algorithm presented in 4.2. We first give some definitions and lemmata needed during the proof.

Definition 2 (Auxiliary Equalities)**.** Let S be a set of variables introduced during purification. By $auxeqs(S)$ we denote the set of all equalities whose left-hand-side is a variable from S:

$$auxeqs(S) = \{v = \phi \mid v \in S\}$$

Given an interpolation problem (A,B), we partition $auxeqs(S)$ into $auxeqs_A(S)$, the conjunction of all auxiliary equalities added to A, and into $auxeqs_B(S)$, the conjunction of those equalities added to B.

Lemma 2 (Auxiliary Equalities and Ordering)**.** Let S be a set of auxiliary variables and v be a minimal element according to inverse dependency order. Then, $v = \phi$ is the only equality containing v in $auxeqs(S)$.

Proof. Suppose $auxeqs(S)$ contains a second equality $v' = \psi(v)$ for some $v' \in S$. According to inverse dependency ordering, v' would be less than v, contradicting our premises. □

Next we prove the interpolant resulting from the algorithm presented in 4.2 correct. The proof relies on $vars$ and $ideps^*$ being closed under set union.

Proof. Every partial interpolant I_C annotating a clause C satisfies following invariants:

$$\begin{aligned} A \wedge auxeqs_A(ideps^*(vars(C))) &\models_{\mathscr{T}} \mathsf{local}_A(C) \vee I_C & (1)\\ B \wedge auxeqs_B(ideps^*(vars(C))) \wedge I_C &\models_{\mathscr{T}} \mathsf{in}_B(C) & (2)\\ syms(I_C) &\subseteq (syms(A) \cap syms(B)) \cup syms(\mathscr{T}) & (3)\\ vars(I_C) &\subseteq ideps^*(vars(C)) & (4)\end{aligned}$$

Correctness of input, auxiliary equality, and instantiation clauses is trivial to show. The intermediate interpolant produced in step 5a, however, does not satisfy these invariants. We provide different invariants and show that the quantifier introduction steps reconstruct a valid partial interpolant. Let QS be the quantification set as defined in step 5b in the algorithm. We use following invariants for step 5c:

$$\begin{aligned} A \wedge auxeqs_A(ideps^*(vars(C))) \wedge auxeqs_A(QS) &\models_{\mathscr{T}} \mathsf{local}_A(C) \vee I_C & (5)\\ B \wedge auxeqs_B(ideps^*(vars(C))) \wedge auxeqs_B(QS) \wedge I_C &\models_{\mathscr{T}} \mathsf{in}_B(C) & (6)\\ syms(I_C) &\subseteq (syms(A) \cap syms(B)) \cup syms(\mathscr{T}) & (7)\\ vars(I_C) &\subseteq ideps^*(vars(C)) \cup QS & (8)\end{aligned}$$

Invariants (5)-(8) hold after 5b. We split cases on the membership of the pivot l of the resolution step.

Case 1: If l occurs in B or $\mathscr{T}$, we know that $l \notin \mathsf{local}_A(C \vee l)$. From premises $A \wedge auxeqs_A(ideps^*(vars(C_1 \vee l))) \models_{\mathscr{T}} \mathsf{local}_A(C_1 \vee l) \vee I_1$ and $A \wedge auxeqs_A(ideps^*(vars(C_2 \vee \neg l))) \models_{\mathscr{T}} \mathsf{local}_A(C_2 \vee \neg l) \vee I_2$ we derive invariant 5 as follows:

Given $A \wedge auxeqs_A(ideps^*(vars(C_1 \vee l))) \wedge auxeqs_A(ideps^*(vars(C_2 \vee \neg l)))$, we know $\mathsf{local}_A(C_1) \vee I_1$ and $\mathsf{local}_A(C_2) \vee I_2$ hold. This implies $\mathsf{local}_A(C_3) \vee (I_1 \wedge I_2)$ holds, too.

Since $vars(C \vee l) = vars(C) \cup vars(l)$, *auxeqs* and *ideps** are closed under set union and $vars(l) = vars(\neg l)$, we can split the premises. By definition of *QS* we know that $ideps^*(vars(C_3)) \cup ideps^*(vars(l)) = ideps^*(vars(C_3)) \cup QS$ holds and we can transform the premises into

$$A \wedge auxeqs_A(ideps^*(vars(C_3))) \wedge auxeqs_A(QS)$$

Hence, we get invariant 5 from premises satisfying 1.

Since $l \in \mathsf{in}_B(C_1 \vee l)$ and $\neg l \in \mathsf{in}_B(C_2 \vee \neg l)$, resolution on l and ordering of the premises yields following formula:

$$B \wedge auxeqs_B(ideps^*(vars(C_3))) \wedge auxeqs_B(ideps^*(vars(l))) \wedge (I_1 \wedge I_2) \models_{\mathscr{T}} \mathsf{in}_B(C_3)$$

We split $ideps^*(vars(l)))$ into *QS* and a part already contained in $ideps^*(vars(C_3))$ and get invariant 6.

Invariant 7 is trivial and 8 holds by definition of *QS*. Hence, $I_3 \equiv I_1 \wedge I_2$ satisfies invariants 5–8.

Case 2: If l does not occur in B or $\mathscr{T}$, we know $l \in \mathsf{local}_A(C \vee l)$ holds. We get invariant 5 by resolution on l and reformulation of the premises like above.

Given $B \wedge auxeqs_B(ideps^*(vars(C_3))) \wedge auxeqs_B(ideps^*(vars(l)))$ we know $I_1 \models_{\mathscr{T}} \mathsf{in}_B(C_1)$ and $I_2 \models_{\mathscr{T}} \mathsf{in}_B(C_2)$ holds. This implies $I_1 \vee I_2 \models_{\mathscr{T}} \mathsf{in}_B(C_3)$. Reformulation of the premises proves invariant 6.

Invariant 7 is trivial and 8 holds by definition of *QS*. Hence, partial interpolant $I_3 \equiv I_1 \vee I_2$ satisfies invariants 5–8.

Step 5a in the algorithm transforms partial interpolants satisfying invariants 1–4 into interpolants satisfying 5–8.

Step 5c preserves invariants (5)-(8). The algorithm iteratively removes a minimal elements v, according to inverse dependency ordering, from inference set *QS*. According to lemma 2, this variable only occurs in the auxiliary equality defining this variable. Additionally it occurs in the partial interpolant I_3.

Case 1: If the equality has been added to A, we have premises

$$\begin{aligned} A \wedge auxeqs_A(ideps^*(vars(C))) \wedge auxeqs_A(QS \setminus \{v\}) \wedge v = \phi \quad &\models_{\mathscr{T}} \quad \mathsf{local}_A(C) \vee I_C(v) && (9)\\ B \wedge auxeqs_B(ideps^*(vars(C))) \wedge auxeqs_B(QS \setminus \{v\}) \wedge I_C(v) \quad &\models_{\mathscr{T}} \quad \mathsf{in}_B(C) && (10)\end{aligned}$$

Since 9 is a $\mathscr{T}$-tautology, it holds for all possible values of v, in particular for $v = \phi$. Hence, the formula

$$A \wedge auxeqs_A(ideps^*(vars(C))) \wedge auxeqs_A(QS \setminus \{v\}) \models_{\mathscr{T}} \mathsf{local}_A(C) \vee \exists v.\ I_C(v)$$

is a $\mathscr{T}$-tautology as well. Hence, $\exists v.\ I_C(v)$ satisfies 5 after removing v from *QS*.

In 10, v occurs only in the partial interpolant. Furthermore, this formula is a $\mathscr{T}$-tautology and, hence, has to hold for every value of v. We get the $\mathscr{T}$-tautology

$$\forall v.\ B \wedge auxeqs_B(ideps^*(vars(C))) \wedge auxeqs_B(QS \setminus \{v\}) \wedge I_C(v) \to \mathsf{in}_B(C)$$

Pushing the quantifier in front of $I_C(v)$ yields

$$B \wedge auxeqs_B(ideps^*(vars(C))) \wedge auxeqs_B(QS \setminus \{v\}) \wedge \exists v.\ I_C(v) \models_{\mathscr{T}} \mathsf{in}_B(C)$$

Hence, $\exists v.\ I_C(v)$ satisfies 6 after removing v from the inference set.

Since we do not introduce new symbols invariant 7 is trivially satisfied. Quantification removes v from $vars(I)$ in the next iteration. Therefore, invariant 8 is also satisfied after removing v from *QS*.

Removing a variable mapped to A from the inference set and adding an existential quantifier yields a partial interpolant still respecting invariants 5–8.

Case 2: If the auxiliary equality has been added to B, we have the premises

$$A \wedge \mathit{auxeqs}_A(\mathit{ideps}^*(\mathit{vars}(C))) \wedge \mathit{auxeqs}_A(QS \setminus \{v\}) \models_{\mathscr{T}} \mathsf{local}_A(C) \vee I_C(v) \quad (11)$$

$$B \wedge \mathit{auxeqs}_B(\mathit{ideps}^*(\mathit{vars}(C))) \wedge \mathit{auxeqs}_B(QS \setminus \{v\}) \wedge v = \phi \wedge I_C(v) \models_{\mathscr{T}} \mathsf{in}_B(C) \quad (12)$$

Variable v in 11 occurs only in I_C. Since this formula holds tautologically in $\mathscr{T}$, it holds for every possible value of v. Adding a universal quantifier and pushing it in front of I_C yields the $\mathscr{T}$-tautology

$$A \wedge \mathit{auxeqs}_A(\mathit{ideps}^*(\mathit{vars}(C))) \wedge \mathit{auxeqs}_A(QS \setminus \{v\}) \models_{\mathscr{T}} \mathsf{local}_A(C) \vee \forall v.\ I_C(v)$$

Hence, $\forall v.\ I_C(v)$ satisfies invariant 5 after removing v from QS.

From 12, we know that

$$B \wedge \mathit{auxeqs}_B(\mathit{ideps}^*(\mathit{vars}(C))) \wedge \mathit{auxeqs}_B(QS \setminus \{v\}) \wedge I_C(v) \rightarrow \mathsf{in}_B(C)$$

holds for at least one value of v, namely $v = \phi$. Adding an existential quantifier in front of the implication and pushing it inwards yields

$$B \wedge \mathit{auxeqs}_B(\mathit{ideps}^*(\mathit{vars}(C))) \wedge \mathit{auxeqs}_B(QS \setminus \{v\}) \wedge \forall v.\ I_C(v) \models_{\mathscr{T}} \mathsf{in}_B(C)$$

Invariant 6 holds, therefore, after removing v from QS.

Invariant 7 holds trivially and invariant 8 holds after removing v from QS since quantification removes v from $\mathit{vars}(I)$.

Hence, we can remove a variable, that is mapped to a B-local term, from QS by putting a universal quantifier in front of the current partial interpolant.

Invariants (5)-(8) imply (1)-(4) on termination. After this procedure terminates, the inference set QS is empty and the partial interpolant satisfies the desired invariants 1–4.

For the empty clause $C_3 \equiv \bot$, we have $\mathit{vars}(C_3) = \emptyset$ and $\mathsf{local}_A(C_3) \equiv \mathsf{in}_B(C_3) \equiv \bot$. Invariants 1–4 show, using above results, that the annotation for the empty clause is a valid Craig interpolant. □