Implementation of Interpolation Algorithms for Software Verification

by

José Abel Castellanos Joo

B.Tech., Universidad de las Américas Puebla, 2015

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Science

The University of New Mexico

Albuquerque, New Mexico

July, 2020

Dedication

To my family.

Acknowledgments

TODO.

Implementation of Interpolation Algorithms for Software Verification

by

José Abel Castellanos Joo

B.Tech., Universidad de las Américas Puebla, 2015

M.S., Computer Science, University of New Mexico, 2020

Abstract

This thesis discusses theoretical aspects and an implementation for the interpolation problem of the following theories: (quantifier-free) equalitiy with uninterpreted functions (EUF), unit two variable per inequality (UTVPI), and the combination of the two previous theories. The interpolation algorithms implemented in this thesis are originally proposed in [24].

Refutational proof-based solutions are the usual approach of many interpolation algorithms [19, 32, 31]. The approach taken in [24] relies on quantifier-elimination heuristics to construct an interpolant using one of the two formulas involved in the interpolation problem. The latter makes possible to study the complexity of the algorithms obtained. On the other hand, the combination method implemented in this thesis uses a Nelson-Oppen framework, thus we still require for this particular situation a refutational proof in order to guide the construction of the interpolant for the combined theory.

The implementation uses Z3 [14] for parsing purposes and satisfiability checking in the combination component of the thesis. Minor modifications were applied to

the Z3's enode data structure in order to label and distinguish formulas efficiently (i.e. distinguish A-part, B-part). Thus, the project can easily be integrated to the Z3 solver to extend its functionality for verification purposes using the Z3 plug-in module.

Contents

List of Figures

Chapter 1

Introduction

Modern society is witness of the impact computer software has done in recent years. The benefits of this massive automation is endless. On the other hand, when software fails, it becomes a catostrophe ranging from economic loss to threats for human life.

Due to strict and ambitious agendas, many software products are shipped with unseen and unintentional bugs which might potentially put at risk people's life like critical systems. Several approaches have been used to improve software quality. However, many of these approaches offer partial coverage or might take an abyssal amount of human effort to provide such solutions. Thus, these approaches cannot be consideral practical. On the other hand, for certain applications these contributions are relevant and their proper use fit such workflows uniquely.

Formal methods aim to bring a unique combination of automation, rigor, and efficiency (whenever effecient algorithms exits for the verification task). This thesis discusses a particular problem in software verification known as *the interpolantion problem* for the theories of the quantifier-free fragment of equality with uninterpreted functions (EUF) and unit two variable per inequality (UTVPI) and their combination. These two theories have been studied extensively and researcher have found

several applications for them.

1.1 Backgroud

An interpolant of a pair of logical formulas (α, β) is a logical formula γ such that α implies γ, $\beta \wedge \gamma$ is logically inconsistent, and γ only has common symbols of α and β. Informally this means that the interpolant 'belongs' to the consequence of α, and 'avoids' being part of the consequence of β. Not surprisingly, this intuition is behind many software verification routines where the first formula models a desirable state/property (termination, correctness) of a computer program, and the second formula models the set of undesirable states (non-termination, errors, crashes, etc) of such software. In Chapter 2, an extensive review of the formal concept is provided.

Eventhough interpolants are not a direct concern in verification problems, these problems are found in the core algorithms of the following two applications:

- Refinement of abstract models: In order to improve coverage and decrease the complexity in verification problems, abstract interpretation has become a proper technique to accomplish the latter together with model checking. Eventhough the methodology provides sound results, it is certainly not complete. Additionally, several abstractions do not capture the semantical meaning of programs due to the *over-approximation* approach. Hence, interpolants are used to strength predicate abstractions by using interpolants constructed from valid traces in the abstract model but not valid in the actual model (*spurious counterexamples*) [10, 30, 22].

- Invariant generation: following the same idea as in the previous case, *if a fix point is obtained in the refinement process*, we can obtain a logical invariant

of computer programs [1] Situations where this happens might be due to the finiteness of possible states in the program. It is worth mentioning that the invariant problem as stated in [23] is undecidable [41, 3].

1.2 Related work

There are several interpolation algorithms for the theories involved in the thesis work. The approaches can be classified into the following categories:

- Proof-based approach: This category relies on the availability of a refutational proof. The interpolant is constructed using a recursive function over the structure of the proof tree. In [31, 32] the author defines an interpolation calculus. This particular approach uses a proof tree produced by the SMT solver Z3 and does not need to modify any of Z3's internal mechanisms. Among the advantages of this approach is that theory combination is given for free since the SMT solver takes care of this problem. On the other hand, Z3's satellities theories are not sufficiently integrated with its proof-producing mechanism. Hence, one can find $\mathcal{T}-lemmas$ as black-boxes which introduces incompleteness in the interpolation calculus. For completeness, these lemmas are solved separately by another interpolantion algorithm for the respective theory. In [45] the authors provide a Nelson-Oppen framework to compute interpolants. For the convex-case, the approach only exchange equalities as required by the Nelson-Oppen framework. For the non-convex case, the authors require a resolution-based refutational proof to compute interpolants using Pudlak's algorithm. The introduction of the class of *equality interpolanting theories* is considered among

[1] The interpolation generation approach discussed can be understood as a *lazy framework* similarly to SAT/SMT algorithms. The former is about the production of interpolants, the latter is for assignments/models respectively. Both *block/learn* the formulas in order to find their results.

the most relevant contributions of the paper by the verification community. This is property about theories which states that if a theory is capable of proving a mixed equality $a = b$ (an equality which contains symbols from the two formulas in the interpolanting problem) then it exists a common term in the language of the theory t (known as the interpolating-term) such that the theory can prove $a = t$ and $t = b$. The property facilities formula-splitting for interpolation purposes. In [19] the authors modify a resolution-based refutational proof by introducing common-terms in the proof in order to produce interpolants in what is called colorable-proofs, which are proof trees which do not contained AB-mixed literals. This is pointed as an improvement to the approach followed in [45] which executes a similar idea but done progressively as the proof-tree is built and does not require the theories solver to be *equality propagating* [2]. However, this results is not generalizable to non-convex theories due to internal constraints.

- Reduction-based approach: This method transforms the interpolation problem into a query for some solver related to the theory. An example of this approach can be found in [42] where the authors use a linear-inequality solver to provide an interpolant fot the theory of linear inequalities over the rational/real numbers ($LIA(\mathbb{Q})/LIA(\mathbb{R})$). Additionally, they integrate the procedure with a *hierarchial reasoning* approach in order to incorporate the signature of theory for (quantifer-free) equalities with uninterpreted functions.

1.3 Outline of the thesis

- Chapter 2 provides an extensive background of fundamentals ideas, definitions and decision procedures used in the thesis work.

[2]The authors in [45] require that the theory solvers keep track of the interpolating-term and propagate this term whenever possible

- Chapters 3, 4, and 5 explain implentation details of the interpolating algorithms for the theories EUF, UTVPI, and their combination respectively. These chapters share the same structure. They start with the algorithms used to solve the interpolation problem, discuss about implementation details including diagrams of the architecture of the implemented system, and show a performance comparision with the iZ3 interpolation tool available in the SMT solver Z3 until version 4.7.0.

1.4 Contributions

The contributions of the thesis can be summarized as follow:

1. Implementation of the interpolation algorithm for the theory EUF proposed in [24].
2. Formulation and implementation of a new procedure for checking unsatisfiability of grounded equations in Horn clauses using a congruence closure algorithm with explanations used in the implementation of item 1.
3. Implementation of the interpolation algorithm for the theory UTVPI proposed in [24].
4. Implementation of the combination procedure for interpolating algorithm proposed in [45] in order to combine the implementations of items 1. and item 3.

Chapter 2

Preliminaries

This chapter discusses basic concepts from first-order logic that are used in the rest of this thesis. We will pay particular attention to their language and semantics since these are fundamental concepts necessary to understand the algorithmic constructions to compute interpolants. For a comprehensive treatment on the topic, the reader is suggested the following references [33, 18].

2.1 First-Order Predicate Logic

2.1.1 Language

A language is a collection of symbols of different sorts equipped with a rule of composition that effectively tells us how to recognize elements that belong to the language [43]. In particular, a first-order language is a language that expresses boolean combinations of predicates using terms (constant symbols and function applications). In mathematical terms,

Definition 2.1.1. *A first-order language (also denoted signature) is a triple* $\langle \mathfrak{C}, \mathfrak{P}, \mathfrak{F} \rangle$

of non-logical symbols where:

- $\mathfrak{C}$ *is a collection of constant symbols*
- $\mathfrak{P}$ *is a collection of n-place predicate symbols*
- $\mathfrak{F}$ *is a collection of n-place function symbols*

including logical symbols like quantifiers(universal ($\forall$)*, existential*($\exists$)*) logical symbols like parenthesis, propositional connectives (implication* ($\rightarrow$)*, conjunction* ($\wedge$)*, disjunction* ($\vee$)*, negation* ($\neg$)*), and a countably number of variables* Vars *(i.e.* $\text{Vars} = \{v_1, v_2, \dots\}$*).*

The rules of composition distinguish two objects, terms *and* formulas*, which are defined recursively as follows:*

- *Any variable symbol or constant symbol is a term.*
- *If* $t_1, \dots, t_n$ *are terms and* f *is an n-ary function symbol, then* $f(t_1, \dots, t_n)$ *is also a term.*
- *If* $t_1, \dots, t_n$ *are terms and* P *is an n-ary predicate symbol, then* $P(t_1, \dots, t_n)$ *is formula.*
- *If* x *is a variable and* ψ, φ *are formulas, then* $\neg\psi, \forall x.\psi, \exists x.\psi$ *and* $\psi \square \varphi$ *are formulas where* $\square \in \{\rightarrow, \wedge, \vee\}$

No other expression in the language can be considered terms nor formulas if such expressions are not obtained by the previous rules.

2.1.2 Semantics

In order to define a notion of truth in a first-order language is necessary to associate for each non-logical symbol (since logical symbols have established semantics from propositional logic) a denotation or mathematical object and an *assignment* to the collection of variables. The two previous components are part of an *structure* [18] (or interpretation [43]) for a first-order language.

Definition 2.1.2. *Given a first-order language $\mathfrak{L}$, an interpretation $\mathfrak{I}$ is a pair $(\mathfrak{A}, \mathfrak{J})$, where $\mathfrak{A}$ is a non-empty domain (set of elements) and $\mathfrak{J}$ is a map that associates to the non-logical symbols from $\mathfrak{L}$ the following elements:*

- $c^{\mathfrak{J}} \in \mathfrak{A}$ *for each* $c \in \mathfrak{C}$
- $f^{\mathfrak{J}} \in \{\mathfrak{A}^n \to \mathfrak{A}\}$ *for each n-ary function symbol* $f \in \mathfrak{F}$
- $P^{\mathfrak{J}} \in \{\mathfrak{A}^n\}$ *for each n-ary predicate symbol* $P \in \mathfrak{P}$

An assignment $s : Vars \to \mathfrak{A}$ is a map between Vars to elements from the domain of the interpretation.

With the definition of interpretation $\mathfrak{I}$ and assignment s, we can recursively define a notion of *satisfiability* (denoted by the symbol $\models_{\mathfrak{J},s}$) as a free extension from atomic predicates (function application of predicates) to general formulas as described in [18]. For the latter, we need to extend the assignment function to all terms in the language.

Definition 2.1.3. *Let $\mathfrak{I} = (\mathfrak{A}, \mathfrak{J})$ be an interpretation and s an assignment for a given language, Let $\bar{s} : Terms \to A$ be defined recursively as follows:*

- $\bar{s}(c) = c^{\mathfrak{J}}$

- $\bar{s}(f(t_1, \ldots, t_n)) = f^{\mathfrak{J}}(\bar{s}(t_1), \ldots, \bar{s}(t_n))$

Notice that the extension of s depends on the interpretation used.

Definition 2.1.4. *Given an interpretation $\mathfrak{I} = (\mathfrak{A}, \mathfrak{J})$, an assignment s, and ψ a formula, we define $\mathfrak{I} \models_s \psi$ (read ψ is satisfiable under interpretation $\mathfrak{I}$ and assignment s) recursively as follows:*

- $\models_{\mathfrak{I},s} P(t_1, \ldots, t_n)$ *if and only if* $P^{\mathfrak{J}}(\bar{s}(t_1), \ldots, \bar{s}(t_n))$
- $\models_{\mathfrak{I},s} \neg\psi$ *if and only if it is not the case that* $\models_{\mathfrak{I},s} \psi$
- $\models_{\mathfrak{I},s} \psi \wedge \varphi$ *if and only if* $\models_{\mathfrak{I},s} \psi$ *and* $\models_{\mathfrak{I},s} \varphi$
- $\models_{\mathfrak{I},s} \psi \vee \varphi$ *if and only if* $\models_{\mathfrak{I},s} \psi$ *or* $\models_{\mathfrak{I},s} \varphi$
- $\models_{\mathfrak{I},s} \psi \to \varphi$ *if and only if* $\models_{\mathfrak{I},s} \neg\psi$ *or* $\models_{\mathfrak{I},s} \varphi$
- $\models_{\mathfrak{I},s} \forall x.\psi$ *if and only if for every* $d \in \mathfrak{A}, \models_{\mathfrak{I},s_{x\mapsto d}} \psi$, *where* $s_{x\mapsto d} : Vars \to \mathfrak{A}$ *is reduct of* s *under* $Vars \setminus \{x\}$ *and* $s_{x\mapsto d}(x) = d$
- $\models_{\mathfrak{I},s} \exists x.\psi$ *if and only if exits* $d \in \mathfrak{A}, \models_{\mathfrak{I},s_{x\mapsto d}} \psi$, *where* $s_{x\mapsto d}$ *is defined as in the previous item.*

If an interpretation and assignment satisfies a formula, then we say that the interpretation and the assignment are a model for the respective formula. A collection of formulas are satified by an interpretation and assignment if these model each formula in the collection.

A formula ψ is said to be a valid *formula of the interpretation $\mathfrak{I}$ when $\models_{\mathfrak{I},s} \psi$ for all possible assignments s.*

Additionally, if all the models $(\mathfrak{I}, s)$ in a language of a collection of formulas Γ satisfy a formula ψ, then we say that Γ logically implies ψ (written $\Gamma \models \psi$). For the latter, ψ is said to be a valid *formula of the model $(\mathfrak{I}, s)$.*

2.2 Mathematical Theories

A theory $\mathcal{T}$ is a collection of formulas that are closed under logical implication, i.e. if $\mathcal{T} \models \psi$ then $\psi \in \mathcal{T}$. This concept is quite relevant for our thesis work since we will focus on two theories, the quantifier-free fragment of the theory of equality with uninterpreted functions (EUF), and the theory of unit two variable per inequality (UTVPI).

For some theories it is enough to provide a collection of formulas (known as the axioms of the theory). For the case of the theories of interest for the thesis, the axiomatization if the following:

2.2.1 Equality with uninterpreted functions

Definition 2.2.1. *Let $\mathfrak{L}_{EUF} = \{\{\}, \{=\}, \{f_1, \ldots, f_n\}\}$ be the language of EUF. The axioms of the theory are:*

- *(Reflexivity)* $\forall x.x = x$
- *(Symmetry)* $\forall x.\forall y.x = y \rightarrow y = x$
- *(Transitivity)* $\forall x.\forall y.\forall z.(x = y \wedge y = z) \rightarrow x = z$
- *(Congruence)* $\forall x_1 \ldots \forall x_n.\forall y_1 \ldots \forall y_n.(x_1 = y_1 \wedge \cdots \wedge x_n = y_n) \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$

We notice that the congruence axiom is not a first-order logic axiom, but rather an axiom-scheme since it is necessary to *instantiate* such axiom for every arity of the function symbols in a given language.

2.2.2 Ordered commutative rings

In order to describe the UTVPI theory we will first introduce the language and theory of an ordered commutative ring.

Definition 2.2.2. *Let* $\mathfrak{L}_{Ord-R} = \{, \{0, 1\}, \{=, \leq\}, \{+, -, *\}, \}$ *be the language of an ordered commutative ring* R*. The axioms of the theory are:*

- $\forall x.\forall y.\forall z.x + (y + z) = (x + y) + z$
- $\forall x.\forall y.x + y = y + x$
- $\forall x.x + 0 = x$
- $\forall x.x + (-x) = 0$
- $\forall x.\forall y.\forall z.x * (y * z) = (x * y) * z$
- $\forall x.x * 1 = x$
- $\forall x.\forall y.x * y = y * x$
- $\forall x.\forall y.\forall z.x * (y + z) = x * y + x * z$
- $\forall x.\forall y.\forall z.(y + z) * x = y * x + z * x$
- $\forall x.\forall y.\forall z.x \leq y \rightarrow x + z \leq y + z$
- $\forall x.\forall y.(0 \leq x \wedge 0 \leq y) \rightarrow 0 \leq x * y.$
- $0 \neq 1 \wedge 0 \leq 1$

Section 2.4 discusses computability aspects for the theories of interest that are relevant for verification.

2.3 Interpolants

Following the notation in [45], we denote $\mathcal{V}(\psi)$ to be the set of non-logical symbols, variables and constants of formula ψ. Given an instance for the interpolation problem (A, B) [1], we distinguish the following categories:

- ψ is *A-local* if $\mathcal{V}(\psi) \in \mathcal{V}(A) \setminus \mathcal{V}(B)$
- ψ is *B-local* if $\mathcal{V}(\psi) \in \mathcal{V}(B) \setminus \mathcal{V}(A)$
- ψ is *AB-common* if $\mathcal{V}(\psi) \in \mathcal{V}(A) \cap \mathcal{V}(B)$
- ψ is *AB-pure* when either $\mathcal{V}(\psi) \subseteq \mathcal{V}(A)$ or $\mathcal{V}(\psi) \subseteq \mathcal{V}(B)$, otherwise ψ is *AB-mixed*

Example 2.3.0.1. *Consider the following interpolation pair:* $(f(a+2)+1 = c+1 \wedge f(a+2) = 0, f(c) \leq b \wedge b < f(0))$. *With respect to the previous interpolation pair, we can tell that:*

- *The formula* $f(a+2) = c$ *is AB-pure but not A-local nor B-local nor AB-common*
- *The formula* $\neg(a \leq f(f(b)+1))$ *is an AB-mixed literal*
- *The formula* $a + 1 = 1$ *is A-local.*
- *The formula* $c + 1 = 1$ *is AB-pure but not AB-common.*
- *The formula* $c = 0$ *is AB-common.*
- *In general,* $AB - common$ *formulas are not* $AB - pure$ *formulas.*

[1] For the rest of the thesis, we will denote the first formula of an interpolantion problem as the A-part and the second component as the B-part

2.3.1 Craig interpolation theorem

Let α, β, γ be logical formulas in a given theory. If $\models_{\mathcal{T}} \alpha \rightarrow \beta$, we say that γ is an interpolant for the interpolation pair (α, β) if the following conditions are met:

- $\models_{\mathcal{T}} \alpha \rightarrow \gamma$
- $\models_{\mathcal{T}} \gamma \rightarrow \beta$
- Every non-logical symbol in γ occurs both in α and β.

The *interpolantion problem* can be stated naturally as follows: given two logical formulas α, β such that $\models_{\mathcal{T}} \alpha \rightarrow \beta$, find the interpolant for the pair (α, β).

In his celebrated result [12], Craig proved that for every pair (α, β) of formulas in first-order logic such that $\models \alpha \rightarrow \beta$, an interpolation formula exists. Nonetheless, there are many logics and theories that this result does not hold [26].

Usually, we see the interpolation problem defined differently in the literature, where it is considered β' to be $\neg\beta$ and the problem requires that the pair (α, β') is mutually contradictory (unsatisfiable). This definition was popularized by McMillan [31]. This shift of attention explains partially the further development in interpolation generation algorithms since many of these relied on SMT solvers that provided refutation proofs in order to (re)construct interpolants for different theories (and their combination) [27, 8, 32].

Relaxed definitions are considered to the interpolation problem when dealing with specific theories [45] in a way that interpreted function can be also part of the interpolant. The latter is justified since otherwise, many interpolation formulas might not exists in different theories or the interpolants obtained might not be relevant (for example, lisp programs). This is formalize as follows:

Definition 2.3.1. *[45] Let $\mathcal{T}$ be a first-order theory of a signature Σ and let $\mathcal{L}$ be the class of quantifier-free Σ formulas. Let $\Sigma_{\mathcal{T}} \subseteq \Sigma$ denote a designated set of interpreted symbols in $\mathcal{T}$. Let A, B be formulas in $\mathcal{L}$ such that $A \wedge B \models_{\mathcal{T}} \bot$. A* theory-specific *interpolant for (A, B) in $\mathcal{T}$ is a formula I in $\mathcal{L}$ such that $A \models_{\mathcal{T}} I$, $B \wedge I \models_{\mathcal{T}} \bot$, and I refers only to AB-common symbols and symbols in $\Sigma_{\mathcal{T}}$.*

Example 2.3.1.1. *In example 2.3.0.1 we can tell $c + 1 = 1$ is not an interpolant simplify because the symbol 1 only appears on the A-part. However, if $\Sigma_{\mathcal{LIA}(\mathbb{Z})}$ contains the interpreted symbols of $LIA(\mathbb{Z})$ (i.e. $+, *, 0, 1, 2, \dots$), then $c+1 = 1$ becomes a* theory-spefic *interpolant.*

Notice that $c = 0$ is an interpolant even if the set of interpreted symbols used for interpolation is empty.

2.4 Decision Procedures

Given a theory $\mathcal{T}$ in a formula ψ in the language of the theory, is it possible to know $\models_{\mathcal{T}} \psi$? The last question is known as the verification of the decision problem for the respective theory $\mathcal{T}$. This question has been studied extensively for many theories of interest [5].

Regarding the decidability of the theories mentioned in Section 2.2, it is known that EUF is undecidable [5], and the theory of ordered commutative rings is undecidable when the structure uses integers as the domain and the semantics of the arithmetical operations [18] (on the other hand, this theory can be decidable if we keep the same structure but use the reals as the domain [18]). Nonetheless, the quantifier-free fragment of EUF and the restriction imposed in the decision problem for the UTVPI theory allow efficient algorithms to decide validity and satisfiability in their respective theories [37, 17, 29].

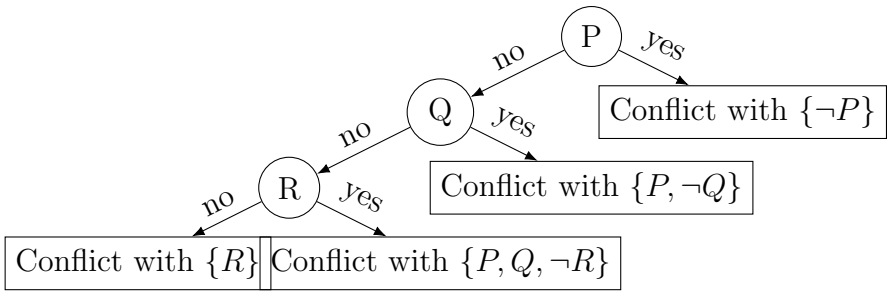


Figure 2.1: Example of DPLL execution on $\{\{\neg P\}, \{P, Q, \neg R\}, \{R\}, \{P, \neg Q\}\}$

In the rest of this section we review some decision problems and provide references to their respective decision procedures used in the implementation work of the thesis.

2.4.1 Satisfiability and Satisfiability Modulo Theories

The satisfiability problem consists on finding a propositional assignment for a propositional formula. This problem is at the core level of complexity theory, defining an important class of problems known as NP, which is a class whose algorithms seem to be intractable. Developments in algorithms and heuristics [25, 35] have made possible to use satisfiability algorithms to solve real-world problems in verification [2].

The DPLL algorithm [13] (an other extensions) is the frequent algorithm found in many SAT solvers. Fundamentally, it is a search-based algorithm which implements of operators (decide, unit-propagation, backtrack) to find a satisfiable assignment. If the algorithm is not able to find a satisfying assignment for a formula, then it is possible to extract a *resolution proof* based on the traces of the search operations.

Example 2.4.0.1. *We can see that the following resolution proof resembles the structure of the DPLL execution on figure 2.1, i.e. if we rotate the proof-tree and mark*

[2]These advances do not provide an answer to the well-known P vs. NP problem. There are results indicating a class of problem instances for many of the SAT algorithms which cannot be solve in less that $\mathcal{O}(2^n)$ steps [25].

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\neg P \quad P \vee Q \vee \neg R}{Q \vee \neg R}\, res_P \qquad R}{Q}\, res_R \qquad P \vee \neg Q}{P}\, res_Q \qquad \neg P}{\bot}\, res_P$$

Figure 2.2: Example of resolution proof

the nodes by the pivots used we obtained a similar tree obtained by the execution of the DPLL algorithm. This fact becomes relevant becomes it enables us to construct a resolution proof from the traces of a SAT solver used in the theory combination part of the implementation.

We can extend this approach to work not only with propositional variables but with terms of more complex signatures [28]. If we are given a boolean combination of formulas from any theory that is capable of deciding the satisfiability of any conjunction of formulas in the theory, by using a *lazy framework* integration with a SAT solver is possible to find either a model or declare unsatisfiable such boolean combination as follows: (i) first abstract the literals in the boolean combination to (pseudo) boolean propositions; (ii) find a satisfying assignment (using a SAT solver) of the (pseudo) boolean propositions; (iii) using the theory solver, test if the collection of positive and negative literals induced by the pseudo boolean variables is satisfiable; (iv) if it is then declare the formula satisfiable, otherwise *learn* (or block) the pseudo boolean clause obtained (by negating the conjunction of boolean constraints) in the SAT solver and repeat from step (ii); (v) if it is not possible to find a satisfying assignment for the pseudo boolean variables, declare the original formula to be unsatisfiable.

These algorithms are used in the last section of the thesis work. The implementation for the interpolation combination method in [45] requires a resolution-based proof in order to compute partial interpolants by integrating Pudlak's algorithm.

2.4.2 Congruence Closure

The congruence closure problem consists of given a conjunction of equalities and disequalities ψ determine if an equality $u = v$ follows from the consequence generated by $\models_{EUF} \psi$.

As noted in [11], it is just succiffient to compute the minimal relation containing the initial relation defined by the equalities in ψ closed under reflexivity, symmetry, transitivity and congruence considering all the subterms in the formulas ψ and $u = v$.

The authors in [17, 37] independently formulated an optimized version of the algorithm afore mentioned. Their key observation was to introduce a list of pointers keeping track of the antecedents nodes in the abstract syntax tree induced by the formulas. The latter allows a fast signature checking in order to determine if two nodes are equivalent under the equivalence relation of the formulas. The algorithm in [17] has better runtime complexity $\mathcal{O}(n \log n)$ since it also implements a 'modify the smaller half' (using the union-find data structure). The congruence closure algorithm in [17] has a runtime complexity of $\mathcal{O}(n^2)$. Nonetheless, the authors reported no significant advantage of the first approach because, for verification purposes, the list of antecedents is usually small. Both approaches provides the FIND, MERGE operations.

In [40], the authors introduced a Union-Find data structure that supports the additional Explanation operation. This operation receives as input an equation between constants. If the input equation is a consequence of the current equivalence relation defined in the Union Find data structure, the Explanation operation returns the minimal sequence of equations used to build such equivalence relation, otherwise it returns 'Not provable'. A proper implementation of this algorithm extends the traditional Union-Find data structure with a *proof-forest*, which consists of an additional representation of the underlying equivalence relation that does not compress

paths whenever a call to the Find operation is made. For efficient reasons, the Find operation uses the path compression and weighted union.

The main observation in [40] is that, in order to recover an explanation between two terms, by traversing the path between the two nodes in the proof tree, the last edge in the path guarantees to be part of the explanation. Intuitively, this follows because only the last Union operation was responsible of merging the two classes into one. Hence, we can recursively recover the rest of the explanation by recursively traversing the subpaths found.

Additionally, the authors in [40] extended the Congruence Closure algorithm [39] using the above data structure to provide Explanations for the theory of EUF. The congruence closure algorithm is a simplification of the congruence closure algorithm in [17]. The latter combines the traditional *pending* and *combine* list into one single list, hence removing the initial *combination* loop in the algorithm in [17].

The implementation work utilizes the latter congruence closure with explanations for the interpolation algorithm of the theory EUF. The idea was to use the explanation operator to construct uncommon-free Horn clauses.

2.4.3 Satisfiability of Horn clauses with ground equations

In [21] it was proposed an algorithm for testing the unsatisfiability of ground Horn clauses with equality. The main idea was to interleave two algorithms: *implicational propagation* (propositional satisfiability of Horn clauses) that updates the truth value of equations in the antecedent of the input Horn clauses [16]; and *equational propagation* (congruence closure for grounded equations) to update the state of a Union-Find data structure [20] that keeps the minimal equivalence relation defined by grounded equations in the input Horn clauses.

The author in [21] defined two variations of his algorithms by adapting the Congruence Closure algorithms in [17, 37]. Additionally, modifications in the data structures used by the original algorithms were needed to make the interleaving mechanism more efficient.

Our implementation uses the equality propagation mechanism in the algorithm proposed by Gallier when we have to deal with Horn clauses with ground equations. In addition, we also needed to design some modifications of the original formulation so it can integrate with the congruence closure with explanation algorithm mentioned in the previous section.

2.4.4 Nelson-Oppen framework for theory and interpolation combination

The theory combination problem consists on taking a formula from the union of two (or more) disjoint languages and tell if such formula is satisfiable or not in the combined theory, i.e. a theory resulting after putting together two (or more) axiomatizations.

In [36] the authors defined a procedure to achieve the above problem. The key idea is to *purify* (or separate) the subformulas by including additional constant symbols equating subterms such that the resulting formula can be splitted into components of the appropriate language for each theory solvers to work with. The separation naturally will hide relevant information to the solvers and they might not be able to decide satisfiability correctly. The authors noticed that to solve the above problem it is enough to share disjunction of equalities between the combined theories of shared terms. In addition, they proved that some theories have the following property:

Definition 2.4.1. *Let $\mathcal{T}$ be a theory. We say that $\mathcal{T}$ is a* convex theory *if a finite conjunction of formulas in $\mathcal{T}$ $\psi = \bigwedge_{i=1}^{m} \psi_i$ satisfies $\psi \models_{\mathcal{T}} \bigvee_{j=1}^{n} x_j = y_j$, then exists*

$k \in \{1, \ldots, n\}$ *such that* $\psi \models_{\mathcal{T}} x_k = y_k$.

Hence, it is important to detect whether the theories involved are convex or not since this can improve performance since convex theories do not need to share disjunctions of equalities as mentioned before (since all these disjunctions imply a single equality).

Example 2.4.1.1.

- *The conjunctive fragment of equality logic is convex since it can always decide the membership of an equation in the equivalence relation.*
- *The theory of UTVPI over the intergers is not convex. To see the latter consider* $1 \leq x \wedge x \leq 2 \models_{UTVPI(\mathbb{Z})} 1 = x \vee 2 = x$. *However, it is not the case that* $1 \leq x \wedge x \leq 2 \models_{UTVPI(\mathbb{Z})} 1 = x$ *nor* $1 \leq x \wedge x \leq 2 \models_{UTVPI(\mathbb{Z})} 2 = x$.

An interpolation combination framework as proposed in [45] follow the same idea towards theory combination. Inductively, they define *partial interpolants* for each shared equality/disjunction of equalities until some theory reaches the unsatisfiable state, which is expected since an interpolant is a pair a mutually contradicting formulas.

This framework was implemented in the thesis work. This framework was chosen in particular since it allows to work with non-convex theories (in our case for the theory of UTVPI over $\mathbb{Z}$).

2.5 General system description

The algorithms implemented in this thesis used the C++ programming language. The overall architecture of the system is the following:

All the decision procedures mentioned in this chapter were implemented with the exception of the SAT/SMT algorithms. For the latter, PicoSAT/TraceCheck[2] and

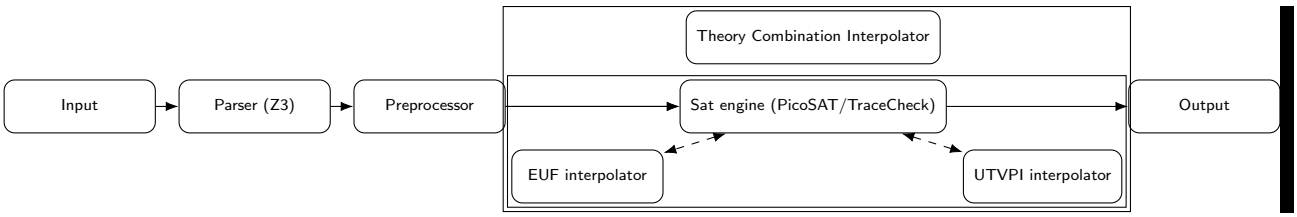


Figure 2.3: General System Diagram

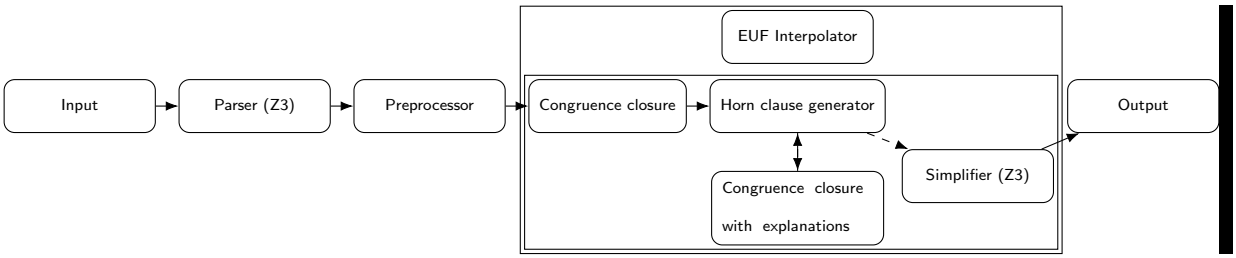


Figure 2.4: EUF Interpolator Diagram

Z3 [14] were chosen as the libraries to work with these algorithms. The rest of this section discusses some minor modifications implemented in the above mentioned Z3 and the proof format output by PicoSAT/TraceCheck.

2.5.1 Minor modifications to Z3

Z3 standard input is SMTLib2 [1]. This grammar does not provide a standard specification regarding a suitable format to work with interpolants. Interpolation

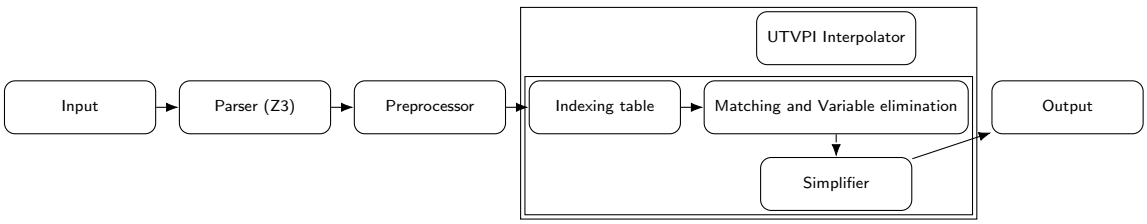


Figure 2.5: UTVPI Interpolator Diagram

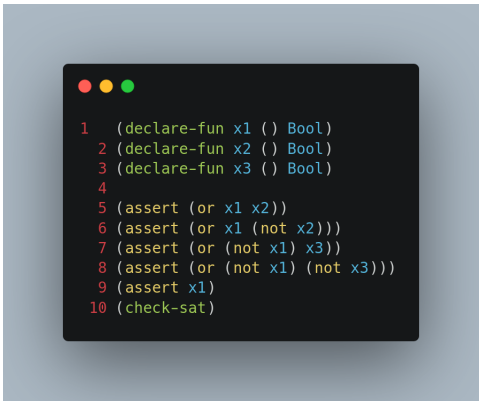
software read interpolant formulas based on the order of appereance in a conjunction [32]. In our case we require two conjuncts of conjunctions of literals in the EUF theory, UTVPI theory or combined theory.

As we can notice in figures 2.3, 2.4, and 2.5, there is preprocessor component which prefixes the names of uninterpreted symbols with the strings a_, b_, c_ to indicate that the symbol name is either an A-local, B-local, or common symbol respectively. We extended Z3's API with functions that test if a formula is A-local, B-local, AB-pure, AB-common based of the definitions in [45] because it is necessary to constantly check this conditions for splitting purposes. Another reason for the latter is justified because the implemented congruence closure algorithm takes as an additional criteria to maintain as representative term an AB-common term. A similar change was implemented in the congruence closure implementation of Z3. Nonetheless, it was irrelevant since Z3's internal structure separates the abstract syntax tree, which is part of its API with the enode data structures, which does not allow the super to modify it. This is the reason why it was not possible to work directly with Z3 congruence closure implementation and a separate implementation was necessary.

2.5.2 PicoSAT/TraceCheck Proof Format

We used PicoSAT/TraceCheck to reconstruct a resolution-based proof necessary for Pudlak's algorithm in the interpolation combination component. Given that Z3 provides a user friendly proof-producing API [15], why did the implementation work require another SAT solver to obtain the resolution-proof?

There are several reasons for the latter. First, the author of the thesis was not able to find an appropriate configuration of parameters for the SMT solver to provide such proofs. In order to grasp an idea of the latter, it was implemented a Z3 proof



```
1  (declare-fun x1 () Bool)
2  (declare-fun x2 () Bool)
3  (declare-fun x3 () Bool)
4
5  (assert (or x1 x2))
6  (assert (or x1 (not x2)))
7  (assert (or (not x1) x3))
8  (assert (or (not x1) (not x3)))
9  (assert x1)
10 (check-sat)
```

Figure 2.6: Problematic SMT query for resolution proofs

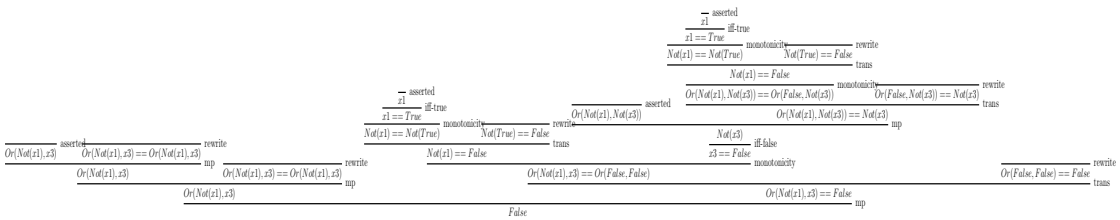


Figure 2.7: Z3 proof of figure 2.6

parser that render a pdf rendered by LaTeX. Many examples indicated that Z3 selects more convenient theories to work with some problems. For instance, the formula in pure propositional logic shown in figure 2.6 used proof rules from EUF [3].

Thus, we opted to use the PicoSAT SAT solver which implements the DPLL algorithm. PicoSAT, as many SAT solvers, take as input a DIMACS cnf file which consists of a straightforward language to denote clauses. (Optional) Lines starting with the character 'c' denote comments, and a single line starting the string 'p cnf' followed by two positive integers denote the number of variables and number of clauses respectively. The next lines encode clauses. Each line is a list of integers numbers terminating with 0. Positive number encode non-negated literals whose identifier is the number, and negative numbers encode the negated literal whose

[3] Z3 uses the term monotonicity instead of congruence

identifier is the absolute value of the number.

The proof format produced by PicoSAT consists of an ordered list of clauses which are either from the input problem (facts) or learned during the search. Each line is encoded following the DIMACS format, but extended to include an identifying number at the beginning of each line (which identifies the clause), and a second list of integers after the terminating zero of the clause which denotes the non-ordered list of clause identifiers involved in a hyper resolution step; the resulting clause obtained from this step is precisely the clauses encoded by the line. An additional zero is used to terminate the second list of integers.

Example 2.5.0.1. *The following extended proof trace is obtained after running the PicoSAT solver on the cnf file encoding the formula in example 2.1:*

$$\begin{array}{cccccc} 1 & -1 & 0 & 0 & & \\ 2 & -3 & 2 & 1 & 0 & 0 \\ 3 & 3 & 0 & 0 & & \\ 4 & 1 & -2 & 0 & 0 & \\ 6 & -2 & 0 & 1 & 4 & 0 \\ 7 & 0 & 1 & 6 & 2 & 3 \end{array}$$

Figure 2.8: Example of extended proof trace by PicoSAT for problem 2.1

As we can notice, the first four lines encode the original input clauses; 1 denotes variable P, 2 denotes variable Q, and 3 denotes variable R. Also, these lines do not include any clause identifier in the second list of number since they are facts. The last two lines denote clauses with identifier 6 and 7. Clause 6 is $\neg Q$, which is obtained after the hyper resolution step using clauses 1 and 4. The last line does not contain elements in the first list since this encodes the contradiction clause.

From the previous example we can observe the list of clause identifiers for the hyper resolution step are not ordered in the sense that we cannot produce multiple resolution steps from the representation. In order to obtain the latter one can

implement a unit propagation algorithm and discover such ordering.

TraceCheck is a tool incorporated in the previous version of PicoSAT [38] to verify proof traces. It is also able to obtain a pure resolution proof. The presentation of the proof trace is restricted to contain only two clause identifiers in the second list for each clause.

Example 2.5.0.2. *The following extended proof trace is obtained after running the TraceCheck on the previous proof trace:*

$$
\begin{array}{cccccccc}
3 & 3 & 0 & 0 & & & & \\
2 & 1 & 2 & -3 & 0 & 0 & & \\
4 & 1 & -2 & 0 & 0 & & & \\
1 & -1 & 0 & 0 & & & & \\
6 & -2 & 0 & 1 & 4 & 0 & & \\
7 & 2 & -3 & 0 & 1 & 2 & 0 & \\
8 & -3 & 0 & 7 & 6 & 0 & & \\
9 & 0 & 8 & 3 & 0 & & &
\end{array}
$$

Figure 2.9: Example of extended proof trace by PicoSAT for problem 2.1

Chapter 3

Interpolation algorithm for the theory of EUF

Interpolation algorithms for the theory of equality with uninterpreted functions are relevant as the core component of verification algorithms. Many useful techniques in software engineering like bounded/unbounded model checking and invariant generation benefit directly from this technique. In [7], the authors introduced a methodology to debug/verify the control logic of pipelined microprocessors by encoding its specification and a logical formula denoting the implementation of the circuit into a EUF solver.

Previous work addressing the interpolation problem for EUF has involved techniques ranging from interpolant-extraction from refutation proof trees [31, 32, 44], and colored congruence closure graphs [19]. Kapur's algorithm uses a different approach by using approximated quantifier-elimination, a procedure that given a formula, it produces a logically equivalent formula without a variable in particular [18].

3.1 Algorithm

Kapur's interpolation algorithm for the EUF theory uses quantifier-elimination techniques to remove symbols in the first formula of an inconsistent pair of formulas that are not common with the second formula of the latter. Hence, the input for this algorithm is a conjunction of equalities in the EUF theory and a set of symbols to eliminate, also known as uncommon symbols. In preparation to discuss Kapur's algorithm we need to provide the following definitions.

Definition 3.1.1. *Let f be a $n-ary$ function symbol and $a_1, \ldots, a_n, b$ terms from the EUF language. We say*

$$f(a_1, \ldots, a_n) = b$$

is an $f-equation$ if the terms $a_1, \ldots, a_n, b$ are constants in the EUF language. We refer to the outermost symbol of the f-equation *as the function symbol appearing in such $f-equation$.*

$f-equations$ are used in Kapur's algorithm to simplify the structure of terms and in Phase II to expose hidden arguments and eliminate uncommon function symbols.

As part of the input of Kapur's algorithm, there is a set of uncommon symbols given directly or computed from the inconsistent pair of formulas by inspection. Using these symbols we can define the following recursive definition of *uncommon terms*:

Definition 3.1.2. *A term t in the EUF language is* uncommon *if:*

- *t is an uncommon constant*
- *t is a function application of the form $f(t_1, \ldots, t_n)$ where either f is an uncommon symbol or any t_i where $1 \leq i \leq n$ is an uncommon term*

Similarly, we can extend homomorphically the notion of uncommon terms to uncommon predicates following a similar construction.

- *Let t_1, t_2 be terms in the EUF language. $t_1 = t_2$, $t_1 \neq t_2$ are uncommon predicates if either t_1 or t_2 are uncommon terms.*
- *Let ψ, φ be predicates in the EUF language. $\psi \star \varphi$ are uncommon predicates if either ψ or φ are uncommon predicates where $\star \in \{\wedge, \vee, \rightarrow\}$. $\neg\psi$ is an uncommon predicate if ψ is an uncommon predicate.*

Regarding extensions of the language with new constants, the *uncommon* property is preserved under equalities. Formally, we mean the following:

Definition 3.1.3. *Let $\mathfrak{L}$ be a EUF language and $\mathfrak{a}$ a constant symbol not belonging to $\mathfrak{L}$. We say $\mathfrak{a}$ is a* common constant under the theory $\mathcal{T}$ in the extended language $\mathfrak{L} \cup \{\mathfrak{a}\}$ *if there exists a common term t in the language $\mathfrak{L}$ such that $\models_{\mathcal{T}} t = \mathfrak{a}$, otherwise $\mathfrak{a}$ is an* uncommon constant*.*

Since congruence closure algorithms are revelant to Kapur's algorithm, we introduce the following definition for notation purposes:

Definition 3.1.4. *Let $\mathcal{E}$ be an equivalence relation between grounded terms of some language $\mathfrak{A}$. The function $repr_{\mathcal{E}} : \mathfrak{A} \to \mathfrak{A}, repr_{\mathcal{E}} : a \mapsto b$ where b is the representative element in $\mathcal{E}$ for a.*

The interpolating formula produced by Kapur's algorith is a conjunction of equations and Horn clauses. A Horn clause if a disjunction of literals which contain at most one non-negated literal. Due to its relevance in the procedure in the EUF theory, for a Horn clause h we denote $antecedent(h)$ to be the conjuntion of disequations in the disjunction of h and $head(h)$ to be either the equation in the disjunction if such is present in h or the particle $\perp$ otherwise.

The main steps in Kapur's algorithm for interpolant generation for the EUF theory are the following:

- **Flattening:** For each subterm t in the input formula assign a fresh unique constant $\mathfrak{a}_t$. Additionally, for each subterm t generate new equations of the form:
  - $c = \mathfrak{a}_c$, if t is a constant c
  - $f(\mathfrak{a}_{t_1}, \ldots, \mathfrak{a}_{t_n}) = \mathfrak{a}_{f(t_1,\ldots,t_n)}$, if t is a function application of the form $f(t_1, \ldots, t_n)$

  Clearly, we can see that this step generates $f - equations$.

- **Elimination of uncommon terms using congruence closure:** This step builds an equivalence relation $\mathcal{E}$ of the $f - equations$ introduced in the Flattening step using a congruence closure algorithm such that the representatives are common terms whenever possible. Uncommon terms appearing in the current conjunction of equations are replaced by their representatives.

- **Horn clause generation by exposure:** This step produces for all pairs of $f - equations$ $(f(\mathfrak{a}_1, \ldots, \mathfrak{a}_n) = \mathfrak{c}, f(\mathfrak{b}_1, \ldots, \mathfrak{b}_n) = \mathfrak{d})$ Horn clauses of the form $\bigwedge_{i=1}^{n}(repr_{\mathcal{E}}(\mathfrak{a}_i) = repr_{\mathcal{E}}(\mathfrak{b}_i)) \rightarrow repr_{\mathcal{E}}(\mathfrak{c}) = repr_{\mathcal{E}}(\mathfrak{d})$ when any of the two following situations happen [1]:
  - The outermost symbol of the $f - equations$ is an uncommon symbol.
  - There is at least one constant argument in any of the $f - equations$ that is an uncommon constant.

[1] Trivial equations in the antecedent of a Horn clause are removed; if the head equation of Horn clause produced in this step is trivial then such Horn clause is discarded

- **Conditional elimination:** We identify the Horn clauses $h := \bigwedge_i (c_i = d_i) \rightarrow a = b$ that have *common antecedents* and uncommon head equations. We perform the following procedure:

  - if a and b are both uncommon terms: replace the equation $a = b$ appearing in the antecedents of all the current Horn clauses by $antecedent(h)$.
  - if either a is common and b uncommon: replace b by a in all the current Horn clauses $h^{'}$ and append $antecedent(h)$ to $antecedent(h^{'})$.
  - if either a is uncommon and b common: Proceed similarly as in the previous case.

  We repeat this step until we cannot produce any new Horn clauses.

- **Conditional replacement:** For each Horn clause of the form $\bigwedge_i (a_i = b_i) \rightarrow u = c$ where the antecedent is common, the term u in its head equation is an uncommon term, and the term c is a common term, replace every instance of u appearing in each $f - equation$ by c to generate Horn clauses with antecedent $\bigwedge_i a_i = b_i$.

  Return the conjunction of formulas obtained as the interpolant.

If the user is not interested in an explicit interpolant, we can present a **lazy/pseudo interpolant** which is an ordered sequence of the original equations together with the Horn clauses produced in Phase II using an appropriate order between the uncommon terms. In order to provide a description of a procedure for the latter we need to introduce the following definitions:

Definition 3.1.5. *Let $\succ$ be a partial order between terms such that $a \succ b$ whenever a is common and b is uncommon. A* dependency pair *for a horn clause $h := \bigwedge_i (a_i = b_i) \rightarrow c = d$ is a pair $(min(c, d, \succ), \{max(c, d, \succ)\} \cup \{u | u$ is an uncommon term*

appering in antecedent(h) }). The first element of a dependency pair is denoted as the target of h and the second element the source of h.

A valid dependency pair *is a dependency pair for some Horn clause h which its target is not included in its source.*

We can notice from definition 3.1.5 than even for equations, its source is never empty.

Definition 3.1.6. *Let $\succ$ be a partial order between terms such that $a \succ b$ whenever a is common and b is uncommon. Given a set of Horn clauses H, a* dependency graph for H *is the directed graph $G_H = (V, E)$ where*

- $V := \{(target_i, source_i) | (target_i, source_i)$ *is a valid depency pair from* $H\}$
- $E := \{(target_i, source_i) \rightarrow (target_j, source_j) | \{target_i\} \cup source_i \subseteq source_j\}$

Using *valid dependency pairs* of the Horn clauses produced in Phase II we can construct an acyclic directed graph as shown by the following theorem:

Theorem 3.1.1. *For any set of Horn clauses H, its dependency graph never contains a cycle between its nodes.*

Proof. Suppose there exists a sequence of n nodes such that $(target_1, source_1) \rightarrow \cdots \rightarrow (target_n, source_n) \rightarrow (target_1, source_1)$. Since $\subseteq$ is transitive we can conclude that $target_1 \in source_1$, which leads to a contradiction since all the nodes in a dependency graph are valid depency pairs. □

Thus, given a set of Horn clauses H, we can compute its dependency graph and use a topological sort algorithm to produce the ordered sequence required in the lazy interpolant representation. Lazy interpolants avoid the possible exponential

size of the formal interpolant. This representation is useful because it provides a more compact representation of the interpolant that the user might be able quicker to obtain. Additionally, the user might be just interested in a particular subformula of the interpolant, so the latter representation offers such feature. This algorithm allows a flexible implementation which can lead several optimizations based on the nature and applications of the interpolant.

3.2 Implementation

The description of the interpolation algorithm presented in the previous section suggests a straight forward implementation of the first two stages using well-known algorithms [37, 17] and data structures from the SMT solver Z3 [14] to represent elements from the EUF language. One particular change was required in the congruence closure algorithm since Kapur's algorithm keeps common terms as representatives whenever common terms belong to a partition of terms induced by the equivalence relation.

The algorithm in [17] uses a union-find data structure to encode the equivalence classes of the nodes in the abstract syntax tree of the input formula with a 'modify the smaller subtree' strategy. This means that when two nodes u, v in the abstract syntax tree are meant to be merged, the representative of the new combined equivalence class is the node which has a bigger numbers of predecessors nodes in the abstract syntax tree pointing to the equivalence class of the node. The idea was to update the least amount of nodes that possibly can change representatives due to the most recent merge operation of the equivalence classes and congruence.

Notation 3.2.1. *Given a theory $\mathcal{E}$ and a term u in the language of $\mathcal{E}$, we can indicate by $[u]$ the equivalence class induced by $\mathcal{E}$, i.e. $[u]_{\mathcal{E}} = \{v \in TERMS | \models_{\mathcal{E}} u = v\}$.* [2]

[2]If the theory is clear from context, the notation $[u]$ denotes the equivalence class of u.

Our algorithm uses a difference partial order to maintain common terms as representatives of the equivalence classes. The non-reflexive relation $\succ_{common}$ [3] is defined for all nodes u, v in the abstract syntax tree of terms as:

$$u \succ_{common} v = \begin{cases} |list(u)| > |list(v)| & \text{if } (u \text{ is a common term} \Leftrightarrow \\ & v \text{ is a common term}) \\ \text{u is a common term} & \text{otherwise} \end{cases}$$

where $list(u) = \{f(u_1, \ldots, u_n) \in \text{TERMS} | \exists i \in \{1, \ldots, n\} \text{ such that } u_i \in [u]]\}$

In the next section we will discuss the changes proposed to Phase III in Kapur's algorithm.

3.2.1 New conditional elimination step in Kapur's algorithm

The modification of Phase III implemented in this thesis work combines and extends the algorithms and data structures introduced in [21, 40]. The algorithm in [21] is a direct extension of [16] which adapts a congruence closure algorithm from [37, 17] in order to update the union-find data structure maintaining the equivalence relation between all the subterms in the input formula. The implementation of the congruence closure algorithm in [40] extends the usual $Find, Merge$ operations on the union-find data structure with the *Explain* operator, which accomplishes the following:

> $Explain(e, e^{'})$: if a sequence U of unions of pairs $(e_1, e_1^{'}), \ldots, (e_p, e_p^{'})$ has taken place, it returns a minimal subset E of U such that $(e, e^{'})$ belongs to the equivalence relationgenerated by E and it returns $\perp$ if no such E exits.

[3] The reflexive relation $u \succeq_{common} v$ is defined as $u = v \vee u \succ_{common} v$, where the equality between nodes is defined as $|list(u)| = |list(v)| \wedge u$ is a common term $\iff v$ is a common term.

The motivation behind the combination of Gallier's data structure and the congruence closure algorithm with explanations is twofold:

1. First, we want to introduce common equations from the antecedents of the Horn clauses obtained during Phase II to a conditional equivalence relation as well as updating the conditional equivalence relation structure by using the equation propagation mechanism from the congruence closure algorithm and the implicational propagation component from Gallier's structure.

2. Additionally, we want to find all the common Horn clauses provable from the original input of equations. The Explain operator in [40] is recursively used to construct the antecedent of such Horn clauses during the conditional replacement step. The Explain operator traverses a proof-tree data structure containing the nodes that were used to combine equivalence classes in the underlying union-find data structure. Thus, the MERGE operation in [21] is required to update the proof tree [4] from [40] as well.

The thesis work accomplishes the previous points by implementing the following:

1. In addition to the parsing procedure and initialization of the Gallier data structure, we assert into the union-find data structure every common equation in the antecedent of a Horn clause.

2. The implemented C++ class for the congruence closure with explanation [5] includes a pointer as data member to the class implementation for the Gallier data structure in order to propagate the equational information achieved during merges and updates due to congruence. The modified algorithms appears below in pseudo-code notation:

[4] This structure is used in order to retrieve explanations.
[5] A fragment of the actual code is shown at Section 6.3.

Algorithm 1 Modified Unsatisfiability Testing for Ground Horn Clauses

```
 1: procedure SATISFIABLE(var H : Hornclause; var queue, combine: queuetype;
    var GT(H) : Graph; var consistent : boolean)
 2:     while queue not empty do
 3:         node := pop(queue);
 4:         for clause_1 in H[node].clauselist do
 5:             if numargs[clause_1] = 0 then
 6:                 nextnode := poslitlist[clause_1];
 7:                 if ¬ H[nextnode].val then
 8:                     if nextnode ≠ ⊥ then
 9:                         queue := push(nextnode, queue);
10:                         H[nextnode].val := true;
11:                         u := left(H[nextnode].atom);
12:                         v := right(H[nextnode].atom);
13:                         if FIND(R, u) ≠ FIND(R, v) then
14:                             combine := push((u, v), combine);
15:                         end if
16:                     else
17:                         consistent := false;
18:                     end if
19:                 end if
20:             end if
21:         end for
22:         if queue is empty then
23:             closure(combine, queue, R);
24:         end if
25:     end while
26: end procedure

27: procedure CLOSURE(var combine, queue : queuetype; var R : partition)
28:     while combine is not empty do
29:         (u, v) = pop(combine)
30:         MERGE(R, u, v, queue)
31:     end while
32: end procedure
```

3. We implement the ExtendedExplain procedure which returns a list of common equations when an equation given as input is provable in the conditional equivalence relation. The pseudo-code for the ExtendedExplain procedure is shown below:

Algorithm 2 Modified Congruence Closure with Explanation Algorithms - Merge

```
    procedure MERGE(R : partition, u, v : node; queue, combine : queuetype)
 2:     if u and v are constants a and b then
            add a = b to Pending;
 4:         Propagate();
        else                                   ▷ u = v is of the form apply(a1, a2) = a
 6:         if Lookup(Representative(a1), Representative(a2)) is some
    apply(b1, b2) = b then
                add (apply(a1, a2) = a, apply(b1, b2) = b) to Pending;
 8:             Propagate();
            else
10:             set Lookup(Representative(a1), Representative(a2)) to
    apply(a1, a2)=a;
                add apply(a1, a2)=a to UseList(Representative(a1)) and to
    UseList(Representative(a2));
12:         end if
        end if
14: end procedure
```

3.2.2 Invariants of the proposed conditional elimination step

We can prove that if a pair of terms t_1, t_2 belong to the same equivalence class, then ExtendedExplain returns a list of common equations using the axiomatization introduced in 2.2.1 as inference rules with the following lemmas:

Lemma 3.2.1. *Let H be a set of Horn clauses produced by Phase II of Kapur's algorithm in some EUF language $\mathcal{L}$, $\mathcal{E}$ the conditional equivalence relation over the terms of $\mathcal{L}$ produced by the above algorithm on H, and t_1, t_2 terms in $\mathcal{L}$ such that $\models_{\mathcal{E}} t_1 = t_2$. If $\exists$ uncommon $a = b \in Explain(t_1, t_2)$, then $\exists h \in H$ such that h is of the form $\bigwedge_i (c_i = d_i) \rightarrow a = b$.*

Proof. By the definition of the Explain operator in 3.2.1, we see that the Explain operator will return a list of all the asserted equations in the conditional equivalence relation. By inspection of the proposed algorithm, assertions into the conditional equivalence relation only happen in two places: when an equation in the antecedent of a horn clause of H is common, and while updating the underlying union-find data

Algorithm 3 Modified Congruence Closure with Explanation Algorithms - Propagate

```
    procedure PROPAGATE( )
 2:     while Pending is non-empty do
            Remove E of the form a=b or (apply(a_1, a_2) = a, apply(b_1, b_2) = b) from
    Pending
 4:         if Representative(a) ≠ Representative(b) and w.l.o.g.
    |ClassList(Representative(a))| ≤ |ClassList(Representative(b))| then
                old_repr_a := Representative(a);
 6:             Insert edge a → b labelled with E into the proof forest;
                for each c in ClassList(old_repr_a) do
 8:                 set Representative(c) to Representative(b)
                    move c from ClassList(old_repr_a) to ClassList(Representative(b))
10:                 for each pointer L in ClassList(u) do
                        if H[L].val = false then
12:                         set the field H[L].lclass or H[L].rclass pointed to by p to
    Representative(b)
                            if H[L].lclass = H[L].rclass then
14:                             queue := push(L, queue);
                                H[L].val := true
16:                         end if
                        end if
18:                 end for
                end for
20:             for each apply(c_1, c_2) = c in UseList(old_repr_a) do
                    if Lookup(Representative(c_1), Representative(c_2)) is some
    apply(d_1, d_2) = d then
22:                     add (apply(c_1, c_2) = c, apply(d_1, d_2) = d) to Pending;
                        remove apply(c_1, c_2) = c from UseList(old_repr_a);
24:                 else
                        set Lookup(Representative(c_1), Representative(c_2)) to apply(c_1,
    c_2) = c;
26:                     move apply(c_1, c_2) = c from UseList(old_repr_a) to
    UseList(Representative(b));
                    end if
28:             end for
            end if
30:     end while
    end procedure
```

structure when the *numargs* field of a Horn clause becomes zero. The latter assert the consequent of such Horn clause. Since u is not common, then such Horn clause should exists in H. □

Algorithm 4 Auxiliary function - ExtendedExplain

procedure EXTENDEDEXPLAIN(t_1 : TERMS, t_2 : TERMS, H : Horn clauses, $\mathcal{E}$: Conditional Equivalence Relation)
2: **if** $Find(t_1, \mathcal{E}) \neq Find(t_2, \mathcal{E})$ **then**
throw Error: t_1, t_2 do not belong to the same equivalence class
4: **end if**
if $t_1.id() = t_2.id()$ **then**
6: **return** {}
end if
8: $\{c, u\}$ = Explain(t_1, t_2, $\mathcal{E}$) where c is a list of common equation and u is a list of uncommon equations
return $c \cup \bigcup\{ExtendedExplain(a, b, H, \mathcal{E}) | consequent \in u, a = b \in antecedent, (\bigwedge antecedent) \rightarrow consequent \in H\}$
10: **end procedure**

Lemma 3.2.2. *Let H be a set of Horn clauses produced by Phase II of Kapur's algorithm in some EUF language $\mathcal{L}$, $\mathcal{E}$ the conditional equivalence relation over the terms of $\mathcal{L}$ produced by the above algorithm on H, and t_1, t_2 terms in $\mathcal{L}$ such that $\models_{\mathcal{E}} t_1 = t_2$.*

The list of equations returned by $ExtendedExplain(t_1, t_2, H, \mathcal{E})$ contains only common equations.

Proof. If t_1, t_2 belong to the same equivalence relation, then there exists a closed derivation (proof tree) with $t_1 = t_2$ as root node and the asserted equations introduced by the modified algorithm as leaves.

The proof proceeds by induction on the complexity of the derivation:

- Case: the last inference rule in the proof tree was Reflexivity. The ExtendedExplain algorithm returns an empty list. Thus, the statement is vacuously true.
- Case: the last inference rule in the proof tree was Symmetry. Line 8 of ExtendedExplain returns a list of common equations c and a list of uncommon equations u. By Lemma 3.2.1 we see that there exists a Horn clause in H for

every uncommon equation in u. Line 9 of ExtendedExplain applies recursively to each of these equations. Since these equations belong to the conditional equivalence relation, there exists a derivation for the latter. By the inductive invariant, ExtendedExplain should return a list of common equation for these subproofs. Thus, the final list contains only common equations.

- Case: the last inference rule in the proof tree was Transitivity. The proof follows similarly to the previous case applied to the two subproofs for the Transitivity rule.
- Case: the last inference rule in the proof tree was Congruence. The proof follwos similarly to the previous case applied n times for each subproof in the Congruence rule.

□

Corollary 3.2.2.1. *Let H be a set of Horn clauses produced by Phase II of Kapur's algorithm, $\mathcal{E}$ the equivalence relation obtained in Phase I of Kapur's algorithm, and $\mathcal{E}'$ the conditional equivalence relation obtained after the modified conditional elimination algorithm. Then $H \models_{\mathcal{E}'} a = b$ if and only if $\exists$ common $x \subseteq \bigcup_{h \in H} antecedent(h)$ such that $H \models_{\mathcal{E}} \bigwedge x \rightarrow a = b$.*

3.2.3 New conditional replacement step in Kapur's algorithm

Once the conditional equivalence relation is built after the execution of the previous step, we can compute conditional eliminations as follows. For the latter, we will require the following auxiliary functions:

By inspection is easy to notice that *Candidates* and *allCandidates* return a set of common terms and a set of sets of common terms respectively.

Algorithm 5 Auxiliary function - Candidates

```
    procedure CANDIDATES(z3::expr const & t)
 2:     if t is common then
            return {t}
 4:     else
            return {t'|t' ∈ Class(t), t' is common}
 6:     end if
    end procedure
```

Algorithm 6 Auxiliary function - allCandidates

```
    procedure ALLCANDIDATES(z3::expr const & t)
 2:     if t is a constant then
            undefined
 4:     end if
        if t has f-symbol uncommon then
 6:         return {{}};
        end if
 8:     if t has f-symbol common and is of the form f(t_1, ..., t_n) then
            return {candidates(t_1), ..., candidates(t_n)};
10:     end if
    end procedure
```

The proposed conditional elimination step produces common Horn clauses from previous uncommon equations and uncommon Horn clauses obtained in Phase II of Kapur's algorithm. The pseudo-code of the algorithms to process the original equations are shown below:

TODO: keep working here. Pay special attention to details of the formalization.

Lemma 3.2.3. *Let* $\{f(\mathfrak{a}_1, \ldots, \mathfrak{a}_n) = \mathfrak{a}\}$ *be the set of equations produced in Phase I of Kapur's algorithms. Executing Conditional elimination only produces common Horn clauses*

TODO: fix vagueness in this paragraph.

In order to process Horn clauses obtained from Phase II of Kapur's algorithm, we apply the conditional elimination step for equation over the equation in the a At the end of this algorithm we can identify *usable Horn clauses* by checking the Horn clauses with *numargs* entries equal to 0.

Algorithm 7 Conditional Elimination - Part 1

```
    procedure CONDITIONAL ELIMINATION(z3::expr const & x, z3::expr const &
    y)
 2:     if x is constant and y is constant then
            for σ_x in Candidates(x) do
 4:             for σ_y in Candidates(y) do
                    horn_clause.add(ExtendedExplain(x, σ_x) + ExtendedExplain(y,
    σ_y), σ_x = σ_y)
 6:             end for
            end for
 8:     end if
        if x is constant and y is of the form f_y(t'_1, ..., t'_k2) then
10:         for σ_x in Candidates(x) do
                for σ_fy in Candidates(f_y(t'_1, ..., t'_k2)) do
12:                 horn_clause.add(ExtendedExplain(x, σ_x) +
    ExtendedExplain(f_y(t'_1, ..., t'_k2), σ_y), σ_x = σ_fy)
                end for
14:             for arguments_fy in CartesianProd(AllCandidates(f_y(t'_1, ..., t'_k2))) do
                    horn_clause.add(ExtendedExplain(x, σ_x) + Σ_{i=1}^{k2}
    ExtendedExplain(t'_i, arguments_fy[i]), σ_x = f_y(arguments_fy))
16:             end for
            end for
18:     end if
        if x is of the form f_x(t_1, ..., t_k1) and y is a constant then
20:         return CONDITIONAL ELIMINATION(y, x);
        end if
22: end procedure
```

The are two main invariants for the Horn clauses formed by the conditional propagation procedure:

- the antecedents are constructed using the *Explain* operator, which returns a sequence of common equations since these are the only merged terms added at the beginning of the initialization routine of the modified Gallier's data structure.

  An additional property must be checked when processing previous Horn clauses. The 'previous' antecedent for these clauses must be *explainable*, which means that every equation in the antecedent must belong to the conditional congruence closure, otherwise an empty explaination will be produced by the *Explain*

Algorithm 8 Conditional Elimination - Part 2

procedure CONDITIONAL ELIMINATION(z3::expr const & x, z3::expr const & y)
  if x is of the form $f_x(t_1, \ldots, t_{k_1})$ and y is of the form $f_y(t_1^{'}, \ldots, t_{k_2}^{'})$ **then**
    for σ_{f_x} in Candidates($f_x(t_1, \ldots, t_{k_1})$) **do**
      for σ_{f_y} in Candidates($f_y(t_1^{'}, \ldots, t_{k_2}^{'})$) **do**
        horn_clause.add(ExtendedExplain($f_x(t_1, \ldots, t_{k_1})$, σ_{f_x}) + ExtendedExplain($f_y(t_1^{'}, \ldots, t_{k_2}^{'})$, σ_y), $\sigma_{f_x} = \sigma_{f_y}$)
      end for
      for $arguments_{f_y}$ in CartesianProd(AllCandidates($f_y(t_1^{'}, \ldots, t_{k_2}^{'})$)) **do**
        horn_clause.add(ExtendedExplain($f_x(t_1, \ldots, t_{k_1})$, σ_{f_x}) + $\sum_{i=1}^{k_2}$ ExtendedExplain($t_i^{'}$, $arguments_{f_y}[i]$), $\sigma_{f_x} = f_y(arguments_{f_y})$)
      end for
    end for
    for $arguments_{f_x}$ in CartesianProd(AllCandidates($f_x(t_1, \ldots, t_{k_1})$)) **do**
      for σ_{f_y} in Candidates($f_y(t_1^{'}, \ldots, t_{k_2}^{'})$) **do**
        horn_clause.add($\sum_{i=1}^{k_1}$ ExtendedExplain(t_i, $arguments_{f_x}[i]$) + ExtendedExplain($f_y(t_1^{'}, \ldots, t_{k_2}^{'})$, σ_y), $f_x(arguments_{f_x}) = \sigma_{f_y}$)
      end for
      for $arguments_{f_y}$ in CartesianProd(AllCandidates($f_y(t_1^{'}, \ldots, t_{k_2}^{'})$)) **do**
        horn_clause.add($\sum_{i=1}^{k_1}$ ExtendedExplain(t_i, $arguments_{f_x}[i]$) + $\sum_{i=1}^{k_2}$ ExtendedExplain($t_i^{'}$, $arguments_{f_y}[i]$), $f_x(arguments_{f_x}) = f_y(arguments_{f_y})$)
      end for
    end for
  end if
end procedure

operator. If a Horn clause antecedent is not *explainable* the resulting Horn clause cannot be added to the final result.

- the consequents are constructed with the help of the *Candidates*, *AllCandidates*, and the *CartesianProd* operators. The former is a function that takes as input a term and returns a list of common terms that belong to the equivalence relation of a given term if such term is uncommon, and return a list containing the term itself if the term is common; the *AllCandidates* function takes as input a function application term and constructs a list of lists with common terms that are equivalent to the arguments of the function application if the

function symbol is common, and returns an empty list otherwise; *CartesianProd* implements a cartesian product, i.e. it produces a list of n-tuples given as input a list of n list of terms (the composition of *CartesianProd* and *AllCandidates* are meant to be used to produce a function application free of uncommon terms). Finally, the head equations for the new Horn clauses are obtained by equating the left-hand side common candidates and right-hand side common candidates obtained by these procedures.

3.2.4 An extra application: Ground Horn Clauses with Explanations

We notice that by adding common equations from the antecedent of the Horn clauses during the conditional elimination step we are able to extend the equivalence relation to a conditional equivalence relation that includes the set of consequences from the afore mentioned common equations. In a more general setting we can specify arbitrarily the set of equations that are meant to enlarge the theory to a conditional equivalence relation for specific purposes.

In particular, given an equivalence relation $\mathcal{E}$, a set of Horn clauses H, and Horn clause h of the form $(a_i = b_i) \rightarrow \mathit{consequent}$, we can provide an explanation to the query $H \models_{\mathcal{E}} h$ by conditionally extending H with the equations from the antecedent of h and then retrieving an explanation for the member of the consequent of H in the conditional equivalence relation.

TODO: keep working here. Fill the missing details.

Lemma 3.2.4. *Let H be blah blah blah Let $\mathit{explanation} := \{\mathit{equation}_1 \ldots \mathit{equation}_n\}$ be the explanation of $H \models_{\mathcal{E}'} \mathit{consequent}$. The explanation for $H \models_{\mathcal{E}} \mathit{antecedent} \rightarrow \mathit{consequent}$ is the subset of explanation that does not contain the equation from antecedent.*

Proof. □

3.3 Evaluation

3.3.1 Detailed evaluation of examples

In this section we discuss in full detail the execution trace of the implementation of some examples.

Symbol elimination example

Let us consider the following example from [24] $\alpha_1 = \{f(z_1, v) = s_1, f(z_2, v) = s_2, f(f(y_1, v), f(y_2, v)) = t\}$ with the set of symbols to eliminate $U_1 = \{u\}$. The implementation produces the following trace (slightly modified for presentation purposes) in order to compute the interpolant for $\alpha_1; U_1$:

```
Before conditionalEliminationEqs
Horn clauses produced
0. 0x5618a6c9c740 (Leader) (= c_y1 (c_f c_y1 a_v)) and (= a_v (c_f c_y2 a_v)) -> (= c_t (c_f c_y1 a_v))
1. 0x5618a6c9c410 (Leader) (= c_z2 (c_f c_y1 a_v)) and (= a_v (c_f c_y2 a_v)) -> (= c_s2 c_t)
2. 0x5618a6c7c6a0 (Leader) (= c_z2 c_y1) -> (= c_s2 (c_f c_y1 a_v))
3. 0x5618a6c4db90 (Leader) (= c_y2 (c_f c_y1 a_v)) and (= a_v (c_f c_y2 a_v)) -> (= c_t (c_f c_y2 a_v))
4. 0x5618a6c9cd30 (Leader) (= a_v (c_f c_y2 a_v)) and (= c_z1 (c_f c_y1 a_v)) -> (= c_s1 c_t)
5. 0x5618a6ca0c10 (Leader) (= c_z1 c_y2) -> (= c_s1 (c_f c_y2 a_v))
6. 0x5618a6ca0b50 (Leader) (= c_z1 c_y1) -> (= c_s1 (c_f c_y1 a_v))
7. 0x5618a6c9c6b0 (Leader) (= c_y1 c_y2) -> (= (c_f c_y1 a_v) (c_f c_y2 a_v))
8. 0x5618a6c7c710 (Leader) (= c_z2 c_y2) -> (= c_s2 (c_f c_y2 a_v))
9. 0x5618a6ca0960 (Leader) (= c_z1 c_z2) -> (= c_s1 c_s2)
Number of horn clauses: 10
Executing conditionalElimination
After conditionalEliminationEqs/Before conditionalEliminationHcs
Horn clauses produced
0. 0x5618a6ca4340 (Leader) (= c_z1 c_y1) and (= c_z1 c_y2) and (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_t (c_f c_s1 c_s2))
1. 0x5618a6ca0960 (Leader) (= c_z1 c_z2) -> (= c_s1 c_s2)
2. 0x5618a6ca0270 (Leader) (= c_z2 c_y1) and (= c_z1 c_y2) and (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_t (c_f c_s2 c_s2))
3. 0x5618a6c7c710 (Leader) (= c_z2 c_y2) -> (= c_s2 (c_f c_y2 a_v))
4. 0x5618a6c9c6b0 (Leader) (= c_y1 c_y2) -> (= (c_f c_y1 a_v) (c_f c_y2 a_v))
5. 0x5618a6ca5b80 (Leader) (= c_z2 c_y1) and (= c_z1 c_y2) -> (= c_t (c_f c_s2 c_s1))
6. 0x5618a6ca0b50 (Leader) (= c_z1 c_y1) -> (= c_s1 (c_f c_y1 a_v))
7. 0x5618a6ca0c10 (Leader) (= c_z1 c_y2) -> (= c_s1 (c_f c_y2 a_v))
```

```
8. 0x5618a6c4db90 (Leader) (= c_y2 (c_f c_y1 a_v)) and (= a_v (c_f c_y2 a_v)) -> (= c_t (c_f c_y2 a_v))
9. 0x5618a6c7c6a0 (Leader) (= c_z2 c_y1) -> (= c_s2 (c_f c_y1 a_v))
10. 0x5618a6ca55a0 (Leader) (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_s1 c_s2)
11. 0x5618a6c9c410 (Leader) (= c_z2 (c_f c_y1 a_v)) and (= a_v (c_f c_y2 a_v)) -> (= c_s2 c_t)
12. 0x5618a6c9cd30 (Leader) (= a_v (c_f c_y2 a_v)) and (= c_z1 (c_f c_y1 a_v)) -> (= c_s1 c_t)
13. 0x5618a6ca62c0 (Leader) (= c_z1 c_y1) and (= c_z1 c_y2) -> (= c_t (c_f c_s1 c_s1))
14. 0x5618a6c9c740 (Leader) (= c_y1 (c_f c_y1 a_v)) and (= a_v (c_f c_y2 a_v)) -> (= c_t (c_f c_y1 a_v))
Number of horn clauses: 15
Executing conditionalEliminationfor Horn clauses
After conditionalEliminationHcs
Horn clauses produced
0. 0x5618a6cabb10 (Leader) (= c_y1 c_y2) and (= c_z2 c_y1) and (= c_z1 c_y2) -> (= c_s1 c_s2)
1. 0x5618a6c9fac0 (Leader) (= c_y1 c_y2) and (= c_z1 c_y1) and (= c_z1 c_y2) and (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_s1 c_s2)
2. 0x5618a6ca4340 (Leader) (= c_z1 c_y1) and (= c_z1 c_y2) and (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_t (c_f c_s1 c_s2))
3. 0x5618a6cab830 (Leader) (= c_z2 c_y1) and (= c_z1 c_y1) -> (= c_s1 c_s2)
4. 0x5618a6ca0960 (Leader) (= c_z1 c_z2) -> (= c_s1 c_s2)
5. 0x5618a6ca0270 (Leader) (= c_z2 c_y1) and (= c_z1 c_y2) and (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_t (c_f c_s2 c_s2))
6. 0x5618a6c7c710 (Leader) (= c_z2 c_y2) -> (= c_s2 (c_f c_y2 a_v))
7. 0x5618a6c9c6b0 (Leader) (= c_y1 c_y2) -> (= (c_f c_y1 a_v) (c_f c_y2 a_v))
8. 0x5618a6ca5b80 (Leader) (= c_z2 c_y1) and (= c_z1 c_y2) -> (= c_t (c_f c_s2 c_s1))
9. 0x5618a6ca0b50 (Leader) (= c_z1 c_y1) -> (= c_s1 (c_f c_y1 a_v))
10. 0x5618a6ca0c10 (Leader) (= c_z1 c_y2) -> (= c_s1 (c_f c_y2 a_v))
11. 0x5618a6c4db90 (Leader) (= c_y2 (c_f c_y1 a_v)) and (= a_v (c_f c_y2 a_v)) -> (= c_t (c_f c_y2 a_v))
12. 0x5618a6ca65b0 (Leader) (= c_z1 c_y2) and (= c_z1 c_y2) and (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_s1 c_s2)
13. 0x5618a6c7c6a0 (Leader) (= c_z2 c_y1) -> (= c_s2 (c_f c_y1 a_v))
14. 0x5618a6ca55a0 (Leader) (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_s1 c_s2)
15. 0x5618a6ca6300 (Leader) (= c_z2 c_y2) and (= c_z1 c_y2) -> (= c_s1 c_s2)
16. 0x5618a6c9c410 (Leader) (= c_z2 (c_f c_y1 a_v)) and (= a_v (c_f c_y2 a_v)) -> (= c_s2 c_t)
17. 0x5618a6c9cd30 (Leader) (= a_v (c_f c_y2 a_v)) and (= c_z1 (c_f c_y1 a_v)) -> (= c_s1 c_t)
18. 0x5618a6ca62c0 (Leader) (= c_z1 c_y1) and (= c_z1 c_y2) -> (= c_t (c_f c_s1 c_s1))
19. 0x5618a6c9c740 (Leader) (= c_y1 (c_f c_y1 a_v)) and (= a_v (c_f c_y2 a_v)) -> (= c_t (c_f c_y1 a_v))
Number of horn clauses: 20
Horn clauses produced
0. 0x5618a6cabb10 (Not leader) (= c_y1 c_y2) and (= c_z2 c_y1) and (= c_z1 c_y2) -> (= c_s1 c_s2)
1. 0x5618a6c9fac0 (Not leader) (= c_y1 c_y2) and (= c_z1 c_y1) and (= c_z1 c_y2) and (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_s1 c_s2)
2. 0x5618a6ca4340 (Leader) (= c_z1 c_y1) and (= c_z1 c_y2) and (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_t (c_f c_s1 c_s2))
3. 0x5618a6cab830 (Not leader) (= c_z2 c_y1) and (= c_z1 c_y1) -> (= c_s1 c_s2)
4. 0x5618a6ca0960 (Leader) (= c_z1 c_z2) -> (= c_s1 c_s2)
5. 0x5618a6ca0270 (Leader) (= c_z2 c_y1) and (= c_z1 c_y2) and (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_t (c_f c_s2 c_s2))
6. 0x5618a6ca5b80 (Leader) (= c_z2 c_y1) and (= c_z1 c_y2) -> (= c_t (c_f c_s2 c_s1))
7. 0x5618a6ca65b0 (Not leader) (= c_z1 c_y2) and (= c_z1 c_y2) and (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_s1 c_s2)
8. 0x5618a6ca55a0 (Not leader) (= c_z1 c_y1) and (= c_z2 c_y1) -> (= c_s1 c_s2)
9. 0x5618a6ca6300 (Not leader) (= c_z2 c_y2) and (= c_z1 c_y2) -> (= c_s1 c_s2)
10. 0x5618a6ca62c0 (Leader) (= c_z1 c_y1) and (= c_z1 c_y2) -> (= c_t (c_f c_s1 c_s1))
Number of horn clauses: 11
Interpolant:
(ast-vector
  (=> (and (= z1 y1) (= z1 y2) (= z1 y1) (= z2 y1)) (= t (f s1 s2)))
  (=> (= z1 z2) (= s1 s2))
  (=> (and (= z2 y1) (= z1 y2) (= z1 y1) (= z2 y1)) (= t (f s2 s2)))
  (=> (and (= z2 y1) (= z1 y2)) (= t (f s2 s1)))
  (=> (and (= z1 y1) (= z1 y2)) (= t (f s1 s1))))
```

The interface offered by SMT solvers with interpolation features usually provide the conventional A-part, B-part format. In order to compare the results with the

implementation two instances were tested so we can obtain interpolants from both systems:

- Problem instance: A-part : $\{f(z_1, v) = s_1, f(z_2, v) = s_2, f(f(y_1, v), f(y_2, v)) = t\}$; B-part : $\{z_1 = z_2, s_1 \neq s_2\}$
  - Z3: (or (= s2 s1) (not (= z2 z1)))
  - Mathsat: (not (and (= z1 z2) (not (= s1 s2))))
  - Our implemenation: ($\rightarrow$ (= z1 z2) (= s1 s2))
- Problem instance: A-part : $\{f(z_1, v) = s_1, f(z_2, v) = s_2, f(f(y_1, v), f(y_2, v)) = t\}$; B-part : $\{z_1 = y_1, z_1 = y_2, f(s_1, s_1) \neq t\}$
  - Z3: (or (= (f s1 s1) t) (not (= z1 y1)) (not (= y2 y1)))
  - Mathsat: (not (and (not (= t (f s1 s1))) (and (= z1 y1) (= z1 y2))))
  - Our implemenation: ($\rightarrow$ (and (= y1 y2) (= z1 y2) (= z1 y2)) (= t (f s1 s1))))

Clearly, the interpolant obtained by just eliminating the symbol v implies the outputs produced by Z3 and Mathsat for the previous problem instances.

Simple example with disequality

Let us consider another example from [24] $\alpha_2 = \{f(x_1) \neq f(x_2)\}$ with the set of symbols to eliminate $U_2 = \{f\}$. The implementation produces the following trace for $\alpha_2; U_2$:

```
Before conditionalEliminationEqs
Horn clauses produced
0. 0x5648882bc710 (Leader) (= c_x2 c_x1) -> (= (a_f c_x2) (a_f c_x1))
1. 0x5648882d7dd0 (Leader) (= (a_f c_x2) (a_f c_x1)) -> false
Number of horn clauses: 2
```

```
Executing conditionalElimination
After conditionalEliminationEqs/Before conditionalEliminationHcs
Horn clauses produced
0. 0x5648882bc710 (Leader) (= c_x2 c_x1) -> (= (a_f c_x2) (a_f c_x1))
1. 0x5648882d7dd0 (Leader) (= (a_f c_x2) (a_f c_x1)) -> false
Number of horn clauses: 2
Executing conditionalEliminationfor Horn clauses
After conditionalEliminationHcs
Horn clauses produced
0. 0x5648882bc710 (Leader) (= c_x2 c_x1) -> (= (a_f c_x2) (a_f c_x1))
1. 0x5648882d7dd0 (Leader) (= (a_f c_x2) (a_f c_x1)) -> false
Number of horn clauses: 2
Horn clauses produced
0. 0x5648882deea0 (Leader) (= c_x2 c_x1) -> false
Number of horn clauses: 1
(ast-vector
  (=> (= x2 x1) false))
```

To compare our result with Z3 and Mathsat we included the B-part formula to be $\{x_1 = x_2\}$. The interpolants obtained by these systems were the same, which was (not (= x1 x2)).

3.4 Conclusions

TODO:

Chapter 4

Interpolation algorithm for UTVPI Formulas

This theory appears heavily in formal methods dealing with abstract domains introduced in [34]. The decision problem consists of checking the satisfiability of a particular fragment of $LIA(\mathbb{Z})$. The fragment consists on conjunctions of inequalities with at most two variables which integers coefficients are restricted to $\{-1, 0, 1\}$. Efficient algorithms are found in the literature for both the satisfiability problem [29] as well as for interpolation [9] of the theory. Eventhough this fragment looks severely constrained, it has been used for a range of applications where problems can be modelled using this particular kind of literals.

The algorithm in [24] follows a similar approach to the interpolation algorithm for EUF in the sense that the attention is given to one of the formulas in the interpolation pair [1] Other approaches towards interpolation follow graph-based algorithms. The idea combines the reduction of UTVPI formulas to difference logic [34] and a cycle detection of maximal size [9].

[1] This implementation uses the first formula of the pair.

4.1 Algorithm

The algorithm proposed [24] uses inference rules to close the relation and eliminate the uncommon variables from the A-part of the input formula. The rules are the following:

$$\frac{s_1 x_1 + s_2 x_2 \leq c \qquad s_1 = s_2 \in \{-1, 0, 1\} \text{ and } x_1 = x_2 \in Vars}{s_1 x_1 \leq \lfloor \frac{c}{2} \rfloor} \text{ Normalize}$$

$$\frac{s_1 x_1 + s_2 x_2 \leq c_1 \qquad -s_2 x_2 + s_3 x_3 \leq c_2}{s_1 x_1 + s_3 x_3 \leq c_1 + c_2} \text{ Elim}$$

The algorithm normalizes the inequalities at the beginning as a preprocessing step and applies the Elim rule whenever it is possible until no more uncommon variables remain in the input formula. Hence, having an efficient representation of the inequalities and detecting matches (like pivots for resolution steps) is important for an efficient implementation. To achieve this goal we implemented to an encoding of the inequalities using natural numbers, an array of numbers indexed by the numeral representation of the inequalities which keeps track of the minimum bound of the encoded inequality, and a data structure to keep track of the signs of variables in the inequalities for efficient matching.

4.2 Implementation

In order to obtain a bijection between UTVPI inequalities and natural numbers, first we define an ordering on the inequalities and notice some invariants of the latter.

We encode the term $\pm x_m \pm x_n$ using the point $(\pm m, \pm n) \in \mathbb{Z}^2$. Let $TermToPoint$ be the map that $\pm x_m \pm x_n \mapsto (\pm m, \pm n)$.The variable x_0 is a *dummy variable* that acts as a place holder for 0. Additionally, we will restrict the terms/pairs such that

the absolute value of the first index variable is strictly greater than the absolute value of the second index variable [2] since addition is commutative, except for the point (0, 0) which encodes the inequality with no variables.

We define the following orderings relevant for the terms of the form $\pm x_m \pm x_n$.

Definition 4.2.1. *Let $\succ_m$ be an ordering on the integers such that $a \succ_m b$ if and only $|a| > |b|$ or ($|a| = |b|$ and $a > b$) where $>$ is the standard ordering on integers.*

Let $\succ_p$ be an ordering on pair of integers such that $(m_1, n_1) \succ_p (m_2, n_2)$ if and only if $m_1 \succ_m m_2$ or ($m_1 = m_2$ and $n_1 \succ_m n_2$).

Let $\succ_t$ be an ordering on terms of the form $\pm x_m \pm x_n$ such that $t_1 \succ_t t_2$ if and only if $TermToPoint(t_1) \succ_p TermToPoint(t_2)$

Example 4.2.1.1. *The first 32 elements (in ascending order w.r.t. $\succ_t$) of UTVPI inequalities [3] are the following:*

$x_0 + x_0 \leq b_0$

$-x_1 + x_0 \leq b_1, x_1 + x_0 \leq b_2$

$-x_2+x_0 \leq b_3, -x_2-x_1 \leq b_4, -x_2+x_1 \leq b_5, x_2+x_0 \leq b_6, x_2-x_1 \leq b_7, x_2+x_1 \leq b_8$

$-x_3 + x_0 \leq b_9, -x_3 - x_1 \leq b_{10}, -x_3 + x_1 \leq b_{11}, -x_3 - x_2 \leq b_{12}, -x_3 + x_2 \leq b_{13}, x_3 + x_0 \leq b_{14}, x_3 - x_1 \leq b_{15}, x_3 + x_1 \leq b_{16}, x_3 - x_2 \leq b_{17}, x_3 + x_2 \leq b_{18}$

From the example, we notice that we can group/order the inequalities by groups using the first index. The first element of the i^{th} group corresponds to the $2(i-1)^2+1$ element in the $\succ_t$ order. The observation follows from an inductive argument since there are $2(1 + 2(i - 1))$ elements in the i^{th} group. It is also straight forward to

[2] This condition also avoids the problem of keeping track of variable duplication in the inequality.

[3] For readability purposes we include the bound for the UTVPI term

find the position of the first element in the second half of any group. With the above information it is possible to find the map between UTVPI terms and naturals numbers.

This bijection allows us to implement a data structure based on a vector of integers extended with $\pm\infty$ which encodes the upper bounds for the i^{th} inequality present in the input formula. For initialization purpose all the entries in this vector are set to ∞ and these values are updated accordingly to keep track to the minimum possible value for the inequality after the application of the inference rules mentioned at the introduction of the section.

4.3 Evaluation

4.3.1 Detailed evaluation of examples

Let us consider the following example: $\alpha_1 = \{x_1 - x_2 \geq -4, -x_2 - x_3 \geq 5, x_2 + x_6 \geq 4, x_2 + x_5 \geq -3\}; \beta_1 = \{-x_1 + x_3 \geq -2, -x_4 - x_6 \geq 0, -x_5 + x_4 \geq 0\}$. Our implementation produces the following trace:

```
Processing
(+ (- c_x1) a_x2)
Updating structure with
x_2 - x_1 <= 4
Processing
(+ a_x2 c_x3)
Updating structure with
x_3 + x_2 <= -5
Processing
(- (- a_x2) c_x6)
Updating structure with
- x_4 - x_2 <= -4
Processing
(- (- a_x2) c_x5)
Updating structure with
- x_5 - x_2 <= 3
Removing this var: x_0
Removing this var: x_1
Removing this var: x_2
```

Algorithm 9 UTVPI constructor

```
    procedure UTVPI CONSTRUCTOR(position : integer)
 2:     coefficient1 = 0
        coefficient2 = 0
 4:     varindex1 = 0
        varindex2 = 0
 6:     if position = 0 then
            return
 8:     end if
        varindex1 = sqrt((position−1)/2) + 1
10:     initial_group_position = 2 ∗ (varindex1 − 1)^2 + 1
        half_size_group = 2 ∗ varindex1 − 1
12:     if position ≤ initial_group_position + half_size_group then
            coefficient1 = -1
14:         if position = initial_group_position then
                coefficient2 = 0
16:             varindex2 = 0
                return
18:         end if
            separation = position - initial_group_position
20:         varindex2 = (separation−1)/2 + 1
            if mod separation 2 = 0 then
22:             coefficient2 = 1
                return
24:         end if
            coefficient2 = -1
26:         return
        end if
28:     coefficient1 = 1
        if position = initial_group_position + half_size_group + 1 then
30:         coefficient2 = 0
            varindex2 = 0
32:         return
        end if
34:     separation = position - initial_group_position - half_size_group - 1
        varindex2 = (separation−1)/2 + 1
36:     if mod separation 2 = 0 then
            coefficient2 = 1
38:         return
        end if
40:     coefficient2 = -1
        return
42: end procedure
```

```
Reducing x_2 - x_1 and - x_4 - x_2
Result: - x_4 - x_1
Reducing x_2 - x_1 and - x_5 - x_2
Result: - x_5 - x_1
Reducing x_3 + x_2 and - x_4 - x_2
Result: - x_4 + x_3
```

Algorithm 10 UTVPI position

```
    procedure UTVPI POSITION( s_1 x_{m_1} + s_2 x_{m_2} : UTVPI term)
 2:     initial_group_position = 2 * (m_1 - 1)^2 + 1
        if s_1 = -1 then
 4:         sign_a_offset = 0
        else
 6:         if s_1 = 0 then
                return 0
 8:         else
                if s_1 = 1 then
10:                 sign_a_offset = 2 * (m_1 - 1) + 1
                end if
12:         end if
        end if
14:     if s_2 = -1 then
            sign_b_offset = 1 + 2 * (m_2 - 1)
16:     else
            if s_2 = 0 then
18:             sign_b_offset = 0
            else
20:             if s_2 = 1 then
                    sign_b_offset = 2 * m_2
22:             end if
            end if
24:     end if
        return initial_group_position + sign_a_offset + sign_b_offset
26: end procedure
```

```
Reducing x_3 + x_2 and - x_5 - x_2
Result: - x_5 + x_3
Removing this var: x_3
Removing this var: x_4
Removing this var: x_5
Interpolant:
(ast-vector
  (<= (- (- x6) x1) 0)
  (<= (+ (- x6) x3) (- 9))
  (<= (- (- x5) x1) 7)
  (<= (+ (- x5) x3) (- 2)))
```

The Z3 SMT solver provides mechanisms to modified the arithmetic engine and several plugins to specialize algorithm for specific theories. It contains a proper specialization to work with Utvpi queries for satisfiability checking. However, the interpolation APIs do not include mechanisms to specialize the interpolation algorithm for Utvpi formulas. Thus, the interpolant obtained by Z3 for the above problem is : (and ($\leq$ 9 (+ (* (- 1) x3) (* 2 x6) x1)) ($\leq$ (- 5) (+ (* (- 1) x3) (* 2 x5) x1))). We can

notice by the coefficients in the result that the interpolant is not a Utvpi formula, thus Z3 must have reduced the problem to linear integer arithmetic.

The result obtained by Mathsat is (and ($\leq$ (- 5) (+ x1 (+ (* (- 1) x3) (* 2 x5)))) ($\leq$ 9 (+ x1 (+ (* (- 1) x3) (* 2 x6))))) which is the same as Z3 modulo the commutativity of the additions in the expression. Despite this difference, the following query to Z3 verifies that the interpolation produced by our implementation implies the interpolation produced by the SMT solver above mentioned; at the same time, the interpolant produced by the SMT solver does not imply our interpolant.

```
(declare-fun x1 () Int)
(declare-fun x3 () Int)
(declare-fun x5 () Int)
(declare-fun x6 () Int)

(define-fun implementation_interpolant () Bool
(and
(<= (- (- x6) x1) 0)
(<= (+ (- x6) x3) (- 9))
(<= (- (- x5) x1) 7)
(<= (+ (- x5) x3) (- 2))
)
)

(define-fun z3_interpolant () Bool
(and
(<= 9 (+ (* (- 1) x3) (* 2 x6) x1))
(<= (- 5) (+ (* (- 1) x3) (* 2 x5) x1))
)
)

(push)
(assert (not (implies  z3_interpolant implementation_interpolant)))
(check-sat) ;; This check returns sat, i.e. z3_interpolant does not imply implementation_interpolant
(pop)
(push)
(assert (not (implies implementation_interpolant z3_interpolant)))
(check-sat) ;; This check returns unsat, i.e. implementation_interpolant implies z3_interpolant
(pop)
```

For the next example, let us consider $\alpha_2 = \{-x_2 - x_1 + 3 \geq 0, x_1 + x_3 + 1 \geq 0, -x_3 - x_4 - 6 \geq 0, x_5 + x_4 + 1 \geq 0\}; \beta_2 = \{x_2 + x_3 + 3 \geq 0, x_6 - x_5 - 1 \geq 0, x_4 - x_6 + 4 \geq 0\}$. Our implementation produces the following trace:

```
Processing
```

```
(+ c_x2 a_x1)
Updating structure with
x_2 + x_1 <= 3
Processing
(- (- c_x3) a_x1)
Updating structure with
- x_3 - x_2 <= 1
Processing
(+ c_x4 c_x3)
Updating structure with
x_4 + x_3 <= -6
Processing
(- (- c_x5) c_x4)
Updating structure with
- x_5 - x_4 <= 1
Removing this var: x_0
Removing this var: x_1
Removing this var: x_2
Reducing x_2 + x_1 and - x_3 - x_2
Result: - x_3 + x_1
Removing this var: x_3
Removing this var: x_4
Removing this var: x_5
(ast-vector
  (<= (+ (- x3) x2) 4)
  (<= (+ x4 x3) (- 6))
  (<= (- (- x5) x4) 1))
```

The interpolant obtained by Z3 is (and (≤ (- 4) (+ x3 (* (- 1) x2))) (≤ (+ x3 x4) (- 6)) (≥ (+ x4 x5) (- 1))); and the interpolant produced by Mathsat is (≤ (+ x2 (+ x3 (+ x4 (* (- 1) x5)))) (- 7)). Using Z3, we were able to verify that:

- the interpolantion obtained by our implementation is equivalent to the interpolant obtained by Z3.
- the interpolant obtained by our implementation implies the interpolant obtained by Mathsat, but the converse does not hold.

```
(declare-fun x1 () Int)
(declare-fun x2 () Int)
(declare-fun x3 () Int)
(declare-fun x4 () Int)
(declare-fun x5 () Int)
(declare-fun x6 () Int)

(define-fun implementation_interpolant () Bool
(and
(<= (+ (- x3) x2) 4)
```

```
(<= (+ x4 x3) (- 6))
(<= (- (- x5) x4) 1))
)

(define-fun z3_interpolant () Bool
(and
(<= (- 4) (+ x3 (* (- 1) x2)))
(<= (+ x3 x4) (- 6))
(>= (+ x4 x5) (- 1)))
)

(define-fun mathsat_interpolant () Bool
(<= (+ x2 (+ x3 (+ x4 (* (- 1) x5)))) (- 7))
)

(push)
(assert (not (iff z3_interpolant implementation_interpolant)))
(check-sat) ;; This check returns unsat, i.e. z3_interpolant is equivalent to implementation_interpolant
(pop)

(push)
(assert (not (implies  mathsat_interpolant implementation_interpolant)))
(check-sat) ;; This check returns sat, i.e. mathsat_interpolant does not imply implementation_interpolant
(pop)

(push)
(assert (not (implies implementation_interpolant mathsat_interpolant)))
(check-sat) ;; This check returns unsat, i.e. implementation_interpolant implies mathsat_interpolant
(pop)
```

4.4 Conclusions

TODO:

Chapter 5

Interpolation algorithm for the theory combination of EUF and UTVPI

Theory combination techniques involve reusing the algorithms for verification problems of some of the theories involved by either purifying terms over the theories, or reducing the original problem into a base theory, or a combination of these two. Usually the approaches following the first approach aforementioned rely on a Nelson-Oppen framework [45, 4, 6]

Despite possibly more efficient approaches using model-based theory combination techniques [1] [4], the latter require operations which many SMT solvers does not provide proper API to implement these. Hence, the approach used in the thesis work follows [45] since it does not require extensive modification to the decision procedures used. Many of the necessary modifications were implemented on top of

[1] An interesting property of model-based theory combination is that it is not needed to propagate disjunctions, even if the theories are not convex. The reason for this is because the *disjunctiviness* of the problem is handled *lazily* by the SAT solver.

Z3 and PicoSAT/TraceCheck.

The propagation of equalities and disjunction of equalities requires the additional step to split these formulas into the correct part of the interpolant pair. Among the major contributions of [45] was the introduction of the class of equality interpolanting theories, which facilites the splitting of a propagated formula since it is clear that A-local/B-local terms should be included in the A-part/B-part of the interpolant pair respectively.

Definition 5.0.1. *A theory $\mathcal{T}$ is* equality interpolanting *if for every a (A-local term) and b (B-local term) and ψ a formula in $\mathcal{T}$ such that $\psi \models_{\mathcal{T}} a = b$, then there exits a term t in $\mathcal{T}$ (called interpolanting term) such that $\psi \models_{\mathcal{T}} a = t$ and $\psi \models_{\mathcal{T}} b = t$*

The relevance of the existance of the interpolating term for a deduced AB-mixed equality becomes relevant in the context of splitting a formula into a suitable A-part and B-part respectively. If t

Deciding where to include AB-common terms to either the A-part or the B-part of the interpolantion pair affects the final result since the interpolant will be *closer* to the A-part or to the B-part respectively. The original paper includes AB-common terms to the B-part. However, the implementation work includes AB-common terms to the A-part since the interpolantion algorithms implemented focus the attention on this part.

5.1 Algorithm

The algorithm implements a Nelson-Oppen framework. In order to integrate interpolants to the latter, the algorithm includes a *partial interpolant* every time a disjunction of equalities (conflict clause) is propagated. These *partial interpolants*

are computed from an unsatisfiability proof obtained by including the negation of the disjunction to the formula using the following definition:

Definition 5.1.1. *[45] Let $\langle A, B\rangle$ be a pair of clause sets such that $A \wedge B \vdash \bot$. Let $\mathcal{T}$ be a proof of unsatisfiability of $A \wedge B$. The propositional formula $p(c)$ for a clause c in $\mathcal{T}$ is defined by induction on the proof structe:*

- *if c is one of the input clauses then*
  - *if $c \in A$, then $p(c) := \bot$*
  - *if $c \in B$, then $p(c) := \top$*
- *otherwise, c is a result of resolution, i.e.* $\dfrac{c_1[= x \vee c_1^{'}] \qquad c_2[= \neg x \vee c_2^{'}]}{c[= c_1^{'} \vee c_2^{'}]}$
  - *if $x \in A$ and $x \notin B$ (x is A-local), then $p(c) := p(c_1) \vee p(c_2)$*
  - *if $x \notin A$ and $x \in B$ (x is B-local), then $p(c) := p(c_1) \wedge p(c_2)$*
  - *otherwise (x is AB-common), then $p(c) := (x \vee p(c_1)) \wedge (\neg x \vee p(c_2))$*

5.2 Implementation

The implementation mantains a map data structure that keeps track of the *partial interpolants*. This ensures that the base case for the above formula $p(c)$ is replaced by previous clauses as required in [45].

Since introducting negations is necessary to compute partial interpolants, we noticed the following interaction with the theories involved in the thesis work:

- EUF case: negations of literals in this theory are just disequalities, which the interpolantion algorithm implemented handles as Horn clauses with a false head term.

- UTVPI case: negations of literals in this theory are either disequalities or strict inequalities. The disequalities are purified an appended to the EUF component, and the strict inequalities are appended as a disjunction of two non-strict inequalities following the axioms of number theory with addition and a total order relation [18].

The main loop of the procedure is shown below:

Algorithm 11 Nelson-Oppen Propagation

```
    procedure NELSON-OPPEN PROPAGATION ( z3::expr_vector const & part_A,
    z3::expr_vector const & part_B )
 2:     T_1, T_2 = Purify(part_A, part_B)
        DisjunctionEqualitiesIterator ψ()
 4:     ψ.init()
        while true do
 6:         if T_1 ⊨_EUF ⊥ then
                return T_1
 8:         end if
            if T_2 ⊨_UTVPI ⊥ then
10:             return T_2
            end if
12:         if T_1 ⊨_EUF ψ.current() then
                if T_2 ⊨_UTVPI ψ.current() then
14:                 continue
                else
16:                 append ψ.current() to T_2
                    ψ.init()
18:             end if
            else
20:             if T_2 ⊨_UTVPI ψ.current() then
                    continue
22:             else
                    append ψ.current() to T_2
24:                 ψ.init()
                end if
26:         end if
            UpdatePartialInterpolant(ψ.current())
28:         ψ.next()
        end while
30: end procedure
```

5.3 Evaluation

5.3.1 Detailed evaluation of a complete example

A simple example

Let us consider the following example: $\alpha = \{f(x_1) = 0, x_1 = a, y_1 \leq a\}; \beta = \{x_1 \leq b, y_1 = b, f(y_1) \neq 0\}$. The implementation produces the following interpolant [2]:

```
(let ((a!1 (and (<= (+ (* (- 1) x1) y1) 0) (<= (+ x1 (* (- 1) y1)) (- 1)))))
(let ((a!2 (or (= x1 y1) a!1 (<= (+ (* (- 1) x1) y1) (- 1)))))
(let ((a!3 (or (not (= (f x1) 0)) (= 0 (f x1)) (and a!2 (not (= x1 y1))))))
      (a!5 (or (= x1 y1) (and a!2 (not (= x1 y1))))))
(let ((a!4 (or (not (= x1 y1)) (= (f y1) 0) (and (= (f x1) 0) a!3))))
  (and a!4 a!5)))))
```

The following SMT query verifies that the previous result obtained is an interpolant of the input formula:

```
(declare-fun x1 () Int)
(declare-fun x2 () Int)
(declare-fun x3 () Int)
(declare-fun y1 () Int)
(declare-fun y2 () Int)
(declare-fun y3 () Int)
(declare-fun a () Int)
(declare-fun b () Int)
(declare-fun f (Int) Int)
(declare-fun g (Int) Int)

(define-fun part_a () Bool
(and
(= (f x1) 0)
(= x1 a)
(<= y1 a)
))

(define-fun part_b () Bool
(and
(<= x1 b)
```

[2] A trace of the execution can be found at the Appendix section since it is considerably large to include it in this section

```
(= y1 b)
(distinct (f y1) 0)
))

(define-fun my_interpolant () Bool

(let ((a!1 (and (<= (+ (* (- 1) x1) y1) 0) (<= (+ x1 (* (- 1) y1)) (- 1)))))
(let ((a!2 (or (= x1 y1) a!1 (<= (+ (* (- 1) x1) y1) (- 1)))))
(let ((a!3 (or (not (= (f x1) 0)) (= 0 (f x1)) (and a!2 (not (= x1 y1)))))
      (a!5 (or (= x1 y1) (and a!2 (not (= x1 y1))))))
(let ((a!4 (or (not (= x1 y1)) (= (f y1) 0) (and (= (f x1) 0) a!3))))
  (and a!4 a!5)))))
)

(push)
;; This returns unsat, which verifies that the input
;; is an inconsistent pair of formulas
(assert (and part_a part_b))
(check-sat)
(pop)
(push)
;; This returns unsat, which verifies that the ouput
;; is implied by the A-part
(assert (not (implies part_a my_interpolant)))
(check-sat)
(pop)
(push)
;; This returns unsat, which verifies that the ouput
;; is inconsistent with the B-part
(assert (and my_interpolant part_b))
(check-sat)
(pop)
```

For this example these are the interpolants reported by Z3 and Mathsat respectively:

- Z3 interpolant: (and ($\geq$ (+ x1 (* (- 1) y1)) 0) (= (f x1) 0))
- Mathsat interpolant: (or (= (f y1) 0) ($\leq$ 1 (+ x1 (* (- 1) y1)))))

The following SMT query verifies the strength relation between the interpolants produced:

```
(declare-fun x1 () Int)
(declare-fun x2 () Int)
(declare-fun x3 () Int)
(declare-fun y1 () Int)
```

```
(declare-fun y2 () Int)
(declare-fun y3 () Int)
(declare-fun a () Int)
(declare-fun b () Int)
(declare-fun f (Int) Int)
(declare-fun g (Int) Int)

(define-fun my_interpolant () Bool

(let ((a!1 (and (<= (+ (* (- 1) x1) y1) 0) (<= (+ x1 (* (- 1) y1)) (- 1)))))
(let ((a!2 (or (= x1 y1) a!1 (<= (+ (* (- 1) x1) y1) (- 1)))))
(let ((a!3 (or (not (= (f x1) 0)) (= 0 (f x1)) (and a!2 (not (= x1 y1))))))
      (a!5 (or (= x1 y1) (and a!2 (not (= x1 y1))))))
(let ((a!4 (or (not (= x1 y1)) (= (f y1) 0) (and (= (f x1) 0) a!3))))
  (and a!4 a!5)))))
)

(define-fun z3_interpolant () Bool
(and (>= (+ x1 (* (- 1) y1)) 0) (= (f x1) 0))
)

(define-fun mathsat_interpolant () Bool
(or (= (f y1) 0) (<= 1 (+ x1 (* (- 1) y1))))
)

(push)
;; The following returns sat, which means
;; that my_interpolant does not imply z3_interpolant
(assert (not (implies my_interpolant z3_interpolant)))
(check-sat)
(pop)
(push)
;; The following returns unsat, which means
;; that z3_interpolant implies my_interpolant
(assert (not (implies z3_interpolant my_interpolant)))
(check-sat)
(pop)
(push)
;; The following returns unsat, which means
;; that my_interpolant implies math_interpolant
(assert (not (implies my_interpolant mathsat_interpolant)))
(check-sat)
(pop)
(push)
;; The following returns sat, which means
;; that mathsat_interpolant does not imply my_interpolant
(assert (not (implies mathsat_interpolant my_interpolant )))
(check-sat)
(pop)
(push)
;; The following returns unsat, which means
;; that z3_interpolant implies math_interpolant
(assert (not (implies z3_interpolant mathsat_interpolant)))
(check-sat)
(pop)
(push)
;; The following returns sat, which means
;; that mathsat_interpolant does not imply z3_interpolant
```

```
(assert (not (implies mathsat_interpolant z3_interpolant )))
(check-sat)
(pop)
```

5.3.2 A parametrized example

Let us consider the following parametrized example: $\alpha = \{1 \leq x, x \leq n\}; \beta = \{f(x) = a, f(1) \neq a, f(2) \neq a, \ldots f(n-1) \neq a, f(n) \neq a\}$ where $n > 1, a$ are fixed integers. We can see that an interpolant for this parametrized input problem is $x = 1 \vee x = 2 \vee \cdots \vee x = n-1 \vee x = n$.

For the case $n = 3, 4, 5$, both the implementation, Z3, and Mathsat were able to compute the expected result discussed above. This particular test was useful to check disjunction of equalities propagation mechanism in the implementation work.

5.3.3 Performance evaluation

The following collection of examples were produced from a reduction algorithm that takes an input formula in the theory of arrays combined with total orders, reducing the formula to EUF and total order. We can use our implementation of theory combination of EUF + UTVPI formulas because given an expression of the form $a \star b$ where $\star$ is either $<, \leq, >, \geq, =, \neq$, the latter is equivalent to simplify$(a - b \star 0)$ using Z3's arithmetic rewriter. The latter is justified formally since integers satisfy the cancellation property over addition.

The cases used in this test followed from branching the disjunctions of the following SMT2 query:

```
(set-info :status unknown)
(declare-sort ElementSort 0)
 (declare-sort ArraySort 0)
 (declare-fun e3 () ElementSort)
(declare-fun rd (ArraySort Int) ElementSort)
```

```
(declare-fun i3 () Int)
(declare-fun a () ArraySort)
(declare-fun i1 () Int)
(declare-fun a1 () ArraySort)
(declare-fun e1 () ElementSort)
(declare-fun b () ArraySort)
(declare-fun c2 () ArraySort)
(declare-fun c1 () ArraySort)
(declare-fun fresh_index_0 () Int)
(declare-fun fresh_index_1 () Int)
(declare-fun i2 () Int)
(define-fun part_a () Bool (and
(= (rd a i3) e3)
(=> (distinct i1 i3) (= (rd a i1) (rd a1 i1)))
(=> (distinct i3 i3) (= (rd a i3) (rd a1 i3)))
(= (rd a1 i1) e1)
(=> (distinct i1 i1) (= (rd a1 i1) (rd b i1)))
(=> (distinct i3 i1) (= (rd a1 i3) (rd b i3)))
(=> (> i1 i1) (= (rd c2 i1) (rd b i1)))
(=> (> i3 i1) (= (rd c2 i3) (rd b i3)))
(=> (= (rd c2 i1) (rd b i1)) (= i1 0))
(=> (> i1 i1) (= (rd c1 i1) (rd a i1)))
(=> (> i3 i1) (= (rd c1 i3) (rd a i3)))
(=> (= (rd c1 i1) (rd a i1)) (= i1 0))
(=> (= (rd c2 fresh_index_0) (rd c1 fresh_index_0)) (= fresh_index_0 0))
(=> (> i1 fresh_index_0) (or (= (rd c2 i1) (rd c1 i1)) (= i1 fresh_index_0)))
(=> (> i3 fresh_index_0) (or (= (rd c2 i3) (rd c1 i3)) (= i3 fresh_index_0)))
(=> (> fresh_index_0 fresh_index_0)
    (or (= (rd c2 fresh_index_0) (rd c1 fresh_index_0))
        (= fresh_index_0 fresh_index_0)))
(=> (distinct fresh_index_0 i3) (= (rd a fresh_index_0) (rd a1 fresh_index_0)))
(=> (distinct fresh_index_0 i1) (= (rd a1 fresh_index_0) (rd b fresh_index_0)))
(=> (> fresh_index_0 i1) (or (= (rd c2 fresh_index_0) (rd b fresh_index_0))))
(=> (> fresh_index_0 fresh_index_0)
    (or (= (rd c2 fresh_index_0) (rd c1 fresh_index_0))))
(=> (> fresh_index_0 i1) (or (= (rd c1 fresh_index_0) (rd a fresh_index_0))))
(>= fresh_index_0 fresh_index_1)
(=> (> fresh_index_0 fresh_index_1)
    (distinct (rd c2 fresh_index_0) (rd c1 fresh_index_0)))
(=> (= fresh_index_0 fresh_index_1) (= fresh_index_0 0))
(=> (= (rd c2 fresh_index_1) (rd c1 fresh_index_1)) (= fresh_index_1 0))
(=> (> i1 fresh_index_1)
    (or (= (rd c2 i1) (rd c1 i1)) (= i1 fresh_index_0) (= i1 fresh_index_1)))
(=> (> i3 fresh_index_1)
    (or (= (rd c2 i3) (rd c1 i3)) (= i3 fresh_index_0) (= i3 fresh_index_1)))
(=> (> fresh_index_0 fresh_index_1)
    (or (= (rd c2 fresh_index_0) (rd c1 fresh_index_0))
        (= fresh_index_0 fresh_index_0)
        (= fresh_index_0 fresh_index_1)))
(=> (> fresh_index_1 fresh_index_1)
    (or (= (rd c2 fresh_index_1) (rd c1 fresh_index_1))
        (= fresh_index_1 fresh_index_0)
        (= fresh_index_1 fresh_index_1)))
(=> (distinct fresh_index_1 i3) (= (rd a fresh_index_1) (rd a1 fresh_index_1)))
(=> (distinct fresh_index_1 i1) (= (rd a1 fresh_index_1) (rd b fresh_index_1)))
(=> (> fresh_index_1 i1) (or (= (rd c2 fresh_index_1) (rd b fresh_index_1))))
(=> (> fresh_index_1 fresh_index_0)
    (or (= (rd c2 fresh_index_1) (rd c1 fresh_index_1))))
```

```
(=> (> fresh_index_1 fresh_index_1)
    (or (= (rd c2 fresh_index_1) (rd c1 fresh_index_1))
        (= fresh_index_1 fresh_index_0)))
(=> (> fresh_index_1 i1) (or (= (rd c1 fresh_index_1) (rd a fresh_index_1))))
(>= i1 0)
(>= i3 0)
(>= fresh_index_0 0)
(>= fresh_index_1 0)
))
(define-fun part_b () Bool (and
(< i1 i2)
(< i2 i3)
(distinct (rd c1 i2) (rd c2 i2))
(=> (= (rd c2 fresh_index_0) (rd c1 fresh_index_0)) (= fresh_index_0 0))
(=> (> i1 fresh_index_0) (or (= (rd c2 i1) (rd c1 i1)) (= i1 fresh_index_0)))
(=> (> i2 fresh_index_0) (or (= (rd c2 i2) (rd c1 i2)) (= i2 fresh_index_0)))
(=> (> i3 fresh_index_0) (or (= (rd c2 i3) (rd c1 i3)) (= i3 fresh_index_0)))
(=> (> fresh_index_0 fresh_index_0)
    (or (= (rd c2 fresh_index_0) (rd c1 fresh_index_0))
        (= fresh_index_0 fresh_index_0)))
(=> (> fresh_index_0 fresh_index_0)
    (or (= (rd c2 fresh_index_0) (rd c1 fresh_index_0))))
(>= fresh_index_0 fresh_index_1)
(=> (> fresh_index_0 fresh_index_1)
    (distinct (rd c2 fresh_index_0) (rd c1 fresh_index_0)))
(=> (= fresh_index_0 fresh_index_1) (= fresh_index_0 0))
(=> (= (rd c2 fresh_index_1) (rd c1 fresh_index_1)) (= fresh_index_1 0))
(=> (> i1 fresh_index_1)
    (or (= (rd c2 i1) (rd c1 i1)) (= i1 fresh_index_0) (= i1 fresh_index_1)))
(=> (> i2 fresh_index_1)
    (or (= (rd c2 i2) (rd c1 i2)) (= i2 fresh_index_0) (= i2 fresh_index_1)))
(=> (> i3 fresh_index_1)
    (or (= (rd c2 i3) (rd c1 i3)) (= i3 fresh_index_0) (= i3 fresh_index_1)))
(=> (> fresh_index_0 fresh_index_1)
    (or (= (rd c2 fresh_index_0) (rd c1 fresh_index_0))
        (= fresh_index_0 fresh_index_0)
        (= fresh_index_0 fresh_index_1)))
(=> (> fresh_index_1 fresh_index_1)
    (or (= (rd c2 fresh_index_1) (rd c1 fresh_index_1))
        (= fresh_index_1 fresh_index_0)
        (= fresh_index_1 fresh_index_1)))
(=> (> fresh_index_1 fresh_index_0)
    (or (= (rd c2 fresh_index_1) (rd c1 fresh_index_1))))
(=> (> fresh_index_1 fresh_index_1)
    (or (= (rd c2 fresh_index_1) (rd c1 fresh_index_1))
        (= fresh_index_1 fresh_index_0)))
(>= i1 0)
(>= i2 0)
(>= i3 0)
(>= fresh_index_0 0)
(>= fresh_index_1 0)
))
(compute-interpolant (interp part_a) part_b)
```

The time shown in next table was measured in microseconds:

Performance evaluation									
Test	Time	Test	Time	Test	Time	Test	Time	Test	Time
#1	8459	#21	8604	#41	7777	#61	7836	#81	8609
#2	7757	#22	8620	#42	7781	#62	7791	#82	8739
#3	7686	#23	8584	#43	7724	#63	7838	#83	8642
#4	8742	#24	8663	#44	8644	#64	8633	#84	8633
#5	7813	#25	7728	#45	8655	#65	7737	#85	7843
#6	7743	#26	7777	#46	8662	#66	7716	#86	7775
#7	7766	#27	7863	#47	8568	#67	7729	#87	7785
#8	8651	#28	8644	#48	8636	#68	8590	#88	8781
#9	8584	#29	7739	#49	7810	#69	8712	#89	7821
#10	8614	#30	7700	#50	7818	#70	8670	#90	7868
#11	8655	#31	7749	#51	7730	#71	8804	#91	7852
#12	8600	#32	8570	#52	8680	#72	8694	#92	9172
#13	7727	#33	8746	#53	7703	#73	7740	#93	8709
#14	7760	#34	8634	#54	7700	#74	7796	#94	8641
#15	7752	#35	8654	#55	7766	#75	7812	#95	8729
#16	8688	#36	8605	#56	8565	#76	8667	#96	8722
#17	7708	#37	7768	#57	8616	#77	7745	#97	7829
#18	7754	#38	7749	#58	8670	#78	7787	#98	7798
#19	7609	#39	7759	#59	8578	#79	7751	#99	7819
#20	8575	#40	8695	#60	8688	#80	8655	#100	8739

5.4 Conclusions

TODO:

Chapter 6

Future Work

Regarding implemenation work, the are several improvements that were not explored in the thesis work since many of them do not change the overal complexity of the implementation, but definitely these consume more resources. These particular improvements are the following:

- Improve hash function for terms: during testing the congruence closure algorithm, it was evident that dealing with a large of terms, collisions happened to the point that some terms were merged since the signature function indicated to do so, but this should not happen. Nonetheless, for a typical verification problem the implementation will not find any problems.
- The space needed to encode curry nodes *can be significally reduced* up to an order of $\mathcal{O}(n)$. The reason for this current allocation schema is that it exists a double bonding effect while performing curryification of all the terms in the arguments of function applications.

Regarding potential extension of the systems, it will interesting to explore a more specific interpolation combination approach as in [42]. The Nelson-Oppen

framework seems to be adequate if the goal is to combine several theories or be as general as possible. The reason why the hierarchical reasoning approach was not implemented was because the authors did not make any claim about the possibility to use non-convex theories. Perhaps, this is not a real limitation of the approach.

Appendix A: additional proofs

6.1 Theorems about Interpolants

6.1.1 Interpolants are closed under conjunction and disjunction

Theorem 6.1.1. *If I_1, I_2 are interpolants of $\langle A, B\rangle$, then $I_1 \wedge I_2, I_1 \vee I_2$ are also interpolants of $\langle A, B\rangle$.*

Proof. We notice that $vars(I_1 \wedge I_2) \subseteq vars(A) \cap vars(B)$ and $vars(I_1 \vee I_2) \subseteq vars(A) \cap vars(B)$, otherwise I_1, I_2 couldn't be interpolants.

Since I_1, I_2 are interpolants, we have that $A \vdash I_1$, $B \wedge I_1 \vdash \bot$ and $A \vdash I_2$, $B \wedge I_2 \vdash \bot$

Here are formal proofs for $A \vdash I_1 \wedge I_2$ and $B \wedge I_1 \wedge I_2 \vdash \bot$:

$$\dfrac{\begin{array}{cc} A & A \\ \vdots & \vdots \\ I_1 & I_2 \end{array}}{I_1 \wedge I_2} \qquad \dfrac{\dfrac{B \wedge I_1 \wedge I_2}{B \wedge I_1}}{\begin{array}{c} \vdots \\ \bot \end{array}}$$

Here are formal proofs for $A \vdash I_1 \vee I_2$ and $B \wedge (I_1 \vee I_2) \vdash \bot$:

$$\dfrac{\begin{array}{c}A\\\vdots\\I_1\end{array}}{I_1 \vee I_2} \qquad \dfrac{\begin{array}{c}B \wedge (I_1 \vee I_2)\\\vdots *\\(B \wedge I_1) \vee (B \wedge I_2)\end{array} \qquad \begin{array}{c}\overline{B \wedge I_1}\\\vdots\\\bot\end{array} \qquad \begin{array}{c}\overline{B \wedge I_2}\\\vdots\\\bot\end{array}}{\bot}$$

Where $*$ is any proof applying the distributivity property of the conjunction symbol over the disjunction symbol.

□

6.1.2 Interpolants distribute conjunctions over disjunctions in the A-part

Theorem 6.1.2. *Let $\mathcal{F}_i$ be a set of formulas and I_i an interpolant for each $\langle A \wedge \mathcal{F}_i, B\rangle$ respectively. Then $\bigvee_i I_i$ is an interpolant for $\langle A \wedge (\bigvee_i \mathcal{F}_i), B\rangle$*

Proof. Let $A_i = A \wedge \mathcal{F}_i$ and $\hat{A} = A \wedge (\bigvee_i \mathcal{F}_i)$.

We see that $vars(I_i) \subseteq vars(A_i) \subseteq vars(\hat{A})$, hence $vars(\bigvee_i I_i) = \bigcup_i vars(I_i) \subseteq \bigcup_i vars(A_i) \subseteq vars(\hat{A})$. Similarly we can prove that $vars(\bigvee_i I_i) \subseteq vars(B)$. Thus, $vars(\bigvee_i I_i) \subseteq vars(A \wedge (\bigvee_i \mathcal{F}_i)) \cap vars(B)$.

To prove $\hat{A} \vdash \bigvee_i I_i$: We notice that $\hat{A} = \bigvee_i A_i$. From the latter and using the generalized version of the disjunction elimination rule in logic, i.e.

$$\dfrac{\alpha_1 \vee \cdots \vee \alpha_n \qquad \begin{array}{c}\overline{\alpha_1}\\\vdots\\I_1\end{array}}{I_1 \vee \cdots \vee I_n} \quad \begin{array}{c}\overline{I_1 \vee \cdots \vee I_n}\end{array} \quad \ldots \quad \begin{array}{c}\overline{\alpha_n}\\\vdots\\I_n\end{array} \quad \begin{array}{c}\overline{I_1 \vee \cdots \vee I_n}\end{array}$$

and distributing disjunctions over conjunctions in $\bigvee_i A_i$ the statement holds.

To prove $B \wedge \bigvee_i I_i \vdash \bot$: Since each $B \wedge I_i \vdash \bot$, by using the generalized version of the disjunction elimination rule from above, the result holds.

Therefore, $\bigvee_i I_i$ is an interpolant for $\langle A \wedge (\bigvee_i \mathcal{F}_i), B \rangle$.

□

Appendix B: additional traces

6.2 Trace for example in theory combination chapter

```
Part a
EUF-component
(= (c_f c_x1) c_oct_1)
(= c_x1 a_a)
Octagon-component
(= 0 c_oct_1)
(<= c_y1 a_a)
Part b
EUF-component
(= c_y1 b_b)
(distinct (c_f c_y1) c_oct_2)
Octagon-component
(<= c_x1 b_b)
(= 0 c_oct_2)
Shared variables
(ast-vector
  c_x1
  a_a
  c_y1
  b_b
  c_oct_1
  c_oct_2)
(= 0 c_oct_1)
(<= c_y1 a_a)
(<= c_x1 b_b)
(= 0 c_oct_2)
(= (c_f c_x1) c_oct_1)
(= c_x1 a_a)
(= c_y1 b_b)
(not (= (c_f c_y1) c_oct_2))

Disjunction implied in EUF: (= c_x1 a_a)
Partial interpolant already computed
```

```
Disjunction implied in EUF: (= c_y1 b_b)
Partial interpolant already computed

Disjunction implied in OCT: (= c_x1 c_y1)
Clause Id: 2 (Fact) Predicate: (<= c_y1 a_a) Interpolant(old): false
Clause Id: 3 (Fact) Predicate: (<= c_x1 b_b) Interpolant(old): true
Clause Id: 5 (Fact) Predicate: (= c_x1 a_a) Interpolant(old): false
Clause Id: 6 (Fact) Predicate: (= c_y1 b_b) Interpolant(old): true
Clause Id: 7 (Fact) Predicate: (not (= c_x1 c_y1)) Interpolant(new): false
Clause Id: 8 (Conflict Clause) Predicate: (or (not (<= c_y1 a_a))
    (not (<= c_x1 b_b))
    (not (= c_x1 a_a))
    (not (= c_y1 b_b))
    (= c_x1 c_y1))
Inside partialInterpolantConflict
Case OCT
(ast-vector
  (<= c_y1 a_a)
  (= c_x1 a_a)
  (not (= c_x1 c_y1)))
(ast-vector
  (<= c_x1 b_b)
  (= c_y1 b_b))
DNF: (let ((a!1 (and (<= (- c_y1 a_a) 0)
                (<= (- c_x1 a_a) 0)
                (<= (+ (- c_x1) a_a) 0)
                (<= (- c_x1 c_y1) (- 1))))
      (a!2 (and (<= (- c_y1 a_a) 0)
                (<= (- c_x1 a_a) 0)
                (<= (+ (- c_x1) a_a) 0)
                (<= (+ (- c_x1) c_y1) (- 1)))))
  (or a!1 a!2))
Input for OctagonInterpolant (ast-vector
  (<= (- c_y1 a_a) 0)
  (<= (- c_x1 a_a) 0)
  (<= (+ (- c_x1) a_a) 0)
  (<= (- c_x1 c_y1) (- 1)))
Input for OctagonInterpolant (ast-vector
  (<= (- c_y1 a_a) 0)
  (<= (- c_x1 a_a) 0)
  (<= (+ (- c_x1) a_a) 0)
  (<= (+ (- c_x1) c_y1) (- 1)))
Theory-specific interpolant: (let ((a!1 (and (<= (+ (- c_x1) c_y1) 0) (<= (- c_x1 c_y1) (- 1))))
      (a!2 (and (<= (+ (- c_x1) c_y1) (- 1)))))
  (or a!1 a!2))
Interpolant for OCT: (let ((a!1 (and (<= (+ (- c_x1) c_y1) 0) (<= (- c_x1 c_y1) (- 1))))
      (a!2 (and (<= (+ (- c_x1) c_y1) (- 1)))))
  (and (or a!1 a!2 false false false) true true))
Interpolant((from conflict)new): (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
  (or a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1))))
Clause Id: 9 (Derived(2,8)) Predicate: (or (not (<= c_x1 b_b)) (not (= c_x1 a_a)) (not (= c_y1 b_b)) (= c_x1 c_y1)) Pivot: (<= c_y1 a_a)
Pivot is A-local
Partial interpolant (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
  (or false a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1))))
Interpolant((from derived)new): (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
```

```
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
  (or a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1))))
Clause Id: 10 (Derived(9,3)) Predicate: (or (= c_x1 c_y1) (not (= c_x1 a_a)) (not (= c_y1 b_b))) Pivot: (<= c_x1 b_b)
Pivot is B-local
Partial interpolant (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!2 (or a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
  (and a!2 true)))
Interpolant((from derived)new): (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
  (or a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1))))
Clause Id: 11 (Derived(10,5)) Predicate: (or (= c_x1 c_y1) (not (= c_y1 b_b))) Pivot: (= c_x1 a_a)
Pivot is A-local
Partial interpolant (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
  (or a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1)) false))
Interpolant((from derived)new): (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
  (or a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1))))
Clause Id: 12 (Derived(11,6)) Predicate: (= c_x1 c_y1) Pivot: (= c_y1 b_b)
Pivot is B-local
Partial interpolant (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!2 (or a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
  (and a!2 true)))
Interpolant((from derived)new): (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
  (or a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1))))
Clause Id: 13 (Derived(12,7)) Predicate: false Pivot: (= c_x1 c_y1)
Pivot is AB-common
Partial interpolant: (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!2 (or (= c_x1 c_y1) a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
  (and a!2 (or (not (= c_x1 c_y1)) false))))
Interpolant((from derived)new): (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!2 (or (= c_x1 c_y1) a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
  (and a!2 (not (= c_x1 c_y1)))))
Final interpolant for conflict clause: (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!2 (or (= c_x1 c_y1) a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
  (and a!2 (not (= c_x1 c_y1)))))

Disjunction implied in OCT: (= c_oct_1 c_oct_2)
Clause Id: 1 (Fact) Predicate: (= 0 c_oct_1) Interpolant(old): false
Clause Id: 4 (Fact) Predicate: (= 0 c_oct_2) Interpolant(old): true
Clause Id: 7 (Fact) Predicate: (not (= c_oct_1 c_oct_2)) Interpolant(new): false
Clause Id: 8 (Conflict Clause) Predicate: (or (not (= 0 c_oct_1)) (not (= 0 c_oct_2)) (= c_oct_1 c_oct_2))
Inside partialInterpolantConflict
Case OCT
-------It was sat!
(ast-vector
  (= 0 c_oct_1)
  (not (= c_oct_1 c_oct_2)))
(ast-vector
  (= 0 c_oct_2))
DNF: (let ((a!1 (and (<= c_oct_1 0)
                (<= (- c_oct_1) 0)
```

```
               (<= (+ (- c_oct_1) c_oct_2) (- 1)))))
  (or (and (<= c_oct_1 0) (<= (- c_oct_1) 0) (<= (- c_oct_1 c_oct_2) (- 1)))
      a!1))
Input for OctagonInterpolant (ast-vector
  (<= c_oct_1 0)
  (<= (- c_oct_1) 0)
  (<= (- c_oct_1 c_oct_2) (- 1)))
Input for OctagonInterpolant (ast-vector
  (<= c_oct_1 0)
  (<= (- c_oct_1) 0)
  (<= (+ (- c_oct_1) c_oct_2) (- 1)))
Theory-specific interpolant: (let ((a!1 (and (<= (- c_oct_1) 0)
               (<= c_oct_1 0)
               (<= (+ (- c_oct_2) c_oct_1) (- 1)))))
  (or a!1
      (and (<= (- c_oct_1) 0) (<= c_oct_1 0) (<= (- c_oct_2 c_oct_1) (- 1)))))
Interpolant for OCT: (let ((a!1 (and (<= (- c_oct_1) 0)
               (<= c_oct_1 0)
               (<= (+ (- c_oct_2) c_oct_1) (- 1)))))
(let ((a!2 (or a!1
               (and (<= (- c_oct_1) 0)
                    (<= c_oct_1 0)
                    (<= (- c_oct_2 c_oct_1) (- 1)))
               false
               false)))
  (and a!2 true)))
Interpolant((from conflict)new): (let ((a!1 (and (>= c_oct_1 0)
               (<= c_oct_1 0)
               (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
               (<= c_oct_1 0)
               (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1)))))
  (or a!1 a!2))
Clause Id: 9 (Derived(1,8)) Predicate: (or (not (= 0 c_oct_2)) (= c_oct_1 c_oct_2)) Pivot: (= 0 c_oct_1)
Pivot is AB-common
Partial interpolant: (let ((a!1 (and (>= c_oct_1 0)
               (<= c_oct_1 0)
               (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
               (<= c_oct_1 0)
               (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1)))))
  (and (or (= 0 c_oct_1) false) (or (not (= 0 c_oct_1)) a!1 a!2)))
Interpolant((from derived)new): (let ((a!1 (and (>= c_oct_1 0)
               (<= c_oct_1 0)
               (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
               (<= c_oct_1 0)
               (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1)))))
  (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2)))
Clause Id: 10 (Derived(9,4)) Predicate: (= c_oct_1 c_oct_2) Pivot: (= 0 c_oct_2)
Pivot is AB-common
Partial interpolant: (let ((a!1 (and (>= c_oct_1 0)
               (<= c_oct_1 0)
               (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
               (<= c_oct_1 0)
               (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1))))
      (a!4 (or (not (not (= 0 c_oct_2))) true)))
```

```
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2))))
  (and (or (not (= 0 c_oct_2)) a!3) a!4)))
Interpolant((from derived)new): (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2))))
  (or (not (= 0 c_oct_2)) a!3)))
Clause Id: 11 (Derived(10,7)) Predicate: false Pivot: (= c_oct_1 c_oct_2)
Pivot is AB-common
Partial interpolant: (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2))))
  (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
       (or (not (= c_oct_1 c_oct_2)) false))))
Interpolant((from derived)new): (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2))))
  (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
       (not (= c_oct_1 c_oct_2)))))
Final interpolant for conflict clause: (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2))))
  (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
       (not (= c_oct_1 c_oct_2)))))
EUF solver found a contradiction
(ast-vector
  (= (c_f c_x1) c_oct_1)
  (= c_x1 a_a)
  (= c_y1 b_b)
  (not (= (c_f c_y1) c_oct_2))
  (= c_x1 c_y1)
  (= c_oct_1 c_oct_2))
Clause Id: 1 (Fact) Predicate: (= (c_f c_x1) c_oct_1) Interpolant(old): false
Clause Id: 4 (Fact) Predicate: (not (= (c_f c_y1) c_oct_2)) Interpolant(old): true
Clause Id: 5 (Fact) Predicate: (= c_x1 c_y1) Interpolant(old): (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!2 (or (= c_x1 c_y1) a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
  (and a!2 (not (= c_x1 c_y1)))))
Clause Id: 6 (Fact) Predicate: (= c_oct_1 c_oct_2) Interpolant(old): (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
```

```
               (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2))))
  (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
       (not (= c_oct_1 c_oct_2)))))
Clause Id: 7 (Conflict Clause) Predicate: (or (not (= (c_f c_x1) c_oct_1))
    (= (c_f c_y1) c_oct_2)
    (not (= c_x1 c_y1))
    (not (= c_oct_1 c_oct_2)))
Inside partialInterpolantConflict
Case EUF
Part a: (ast-vector
  (= (c_f c_x1) c_oct_1)
  (= c_x1 c_y1)
  (= c_oct_1 c_oct_2))
Part b: (ast-vector
  (not (= (c_f c_y1) c_oct_2)))
Theory-specific Interpolant for EUF: (and (= c_oct_1 (c_f c_x1)))
Interpolant for EUF: (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
               (<= (+ c_x1 (* (- 1) c_y1)) (- 1))))
      (a!3 (and (>= c_oct_1 0)
               (<= c_oct_1 0)
               (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!4 (and (>= c_oct_1 0)
               (<= c_oct_1 0)
               (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1)))))
(let ((a!2 (or (= c_x1 c_y1) a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1))))
      (a!5 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!3 a!4))))
(let ((a!6 (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!5)
               (not (= c_oct_1 c_oct_2)))))
(let ((a!7 (or (and (= c_oct_1 (c_f c_x1)))
              false
              (and a!2 (not (= c_x1 c_y1)))
              a!6)))
  (and a!7 true)))))
Interpolant((from conflict)new): (let ((a!1 (and (>= c_oct_1 0)
               (<= c_oct_1 0)
               (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
               (<= c_oct_1 0)
               (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1))))
      (a!5 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
               (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2)))
      (a!6 (or (= c_x1 c_y1) a!5 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
(let ((a!4 (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
               (not (= c_oct_1 c_oct_2)))))
  (or a!4 (= c_oct_1 (c_f c_x1)) (and a!6 (not (= c_x1 c_y1)))))))
Clause Id: 8 (Derived(1,7)) Predicate: (or (= (c_f c_y1) c_oct_2) (not (= c_x1 c_y1)) (not (= c_oct_1 c_oct_2))) Pivot: (= (c_f c_x1) c_oct_1)
Pivot is AB-common
Partial interpolant: (let ((a!1 (and (>= c_oct_1 0)
               (<= c_oct_1 0)
               (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
               (<= c_oct_1 0)
               (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1))))
      (a!5 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
               (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2)))
```

```
      (a!6 (or (= c_x1 c_y1) a!5 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
(let ((a!4 (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
                (not (= c_oct_1 c_oct_2)))))
(let ((a!7 (or (not (= (c_f c_x1) c_oct_1))
               a!4
               (= c_oct_1 (c_f c_x1))
               (and a!6 (not (= c_x1 c_y1))))))
  (and (or (= (c_f c_x1) c_oct_1) false) a!7)))))
Interpolant((from derived)new): (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1))))
      (a!5 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2)))
      (a!6 (or (= c_x1 c_y1) a!5 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
(let ((a!4 (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
                (not (= c_oct_1 c_oct_2)))))
(let ((a!7 (or a!4
               (not (= (c_f c_x1) c_oct_1))
               (= c_oct_1 (c_f c_x1))
               (and a!6 (not (= c_x1 c_y1))))))
  (and (= (c_f c_x1) c_oct_1) a!7)))))
Clause Id: 9 (Derived(8,4)) Predicate: (or (not (= c_oct_1 c_oct_2)) (not (= c_x1 c_y1))) Pivot: (= (c_f c_y1) c_oct_2)
Pivot is AB-common
Partial interpolant: (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1))))
      (a!5 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1))))
      (a!9 (or (not (= (c_f c_y1) c_oct_2)) true)))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2)))
      (a!6 (or (= c_x1 c_y1) a!5 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
(let ((a!4 (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
                (not (= c_oct_1 c_oct_2)))))
(let ((a!7 (or a!4
               (not (= (c_f c_x1) c_oct_1))
               (= c_oct_1 (c_f c_x1))
               (and a!6 (not (= c_x1 c_y1))))))
(let ((a!8 (or (= (c_f c_y1) c_oct_2) (and (= (c_f c_x1) c_oct_1) a!7))))
  (and a!8 a!9))))))
Interpolant((from derived)new): (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1))))
      (a!5 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2)))
      (a!6 (or (= c_x1 c_y1) a!5 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
(let ((a!4 (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
                (not (= c_oct_1 c_oct_2)))))
```

```
(let ((a!7 (or a!4
               (not (= (c_f c_x1) c_oct_1))
               (= c_oct_1 (c_f c_x1))
               (and a!6 (not (= c_x1 c_y1))))))
  (or (= (c_f c_y1) c_oct_2) (and (= (c_f c_x1) c_oct_1) a!7))))))
Clause Id: 10 (Derived(9,5)) Predicate: (not (= c_oct_1 c_oct_2)) Pivot: (= c_x1 c_y1)
Pivot is AB-common
Partial interpolant: (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1))))
      (a!5 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2)))
      (a!6 (or (= c_x1 c_y1) a!5 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
(let ((a!4 (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
                (not (= c_oct_1 c_oct_2))))
      (a!9 (or (not (not (= c_x1 c_y1))) (and a!6 (not (= c_x1 c_y1))))))
(let ((a!7 (or a!4
               (not (= (c_f c_x1) c_oct_1))
               (= c_oct_1 (c_f c_x1))
               (and a!6 (not (= c_x1 c_y1))))))
(let ((a!8 (or (not (= c_x1 c_y1))
               (= (c_f c_y1) c_oct_2)
               (and (= (c_f c_x1) c_oct_1) a!7))))
  (and a!8 a!9))))))
Interpolant((from derived)new): (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1))))
      (a!5 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2)))
      (a!6 (or (= c_x1 c_y1) a!5 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
(let ((a!4 (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
                (not (= c_oct_1 c_oct_2))))
      (a!9 (or (= c_x1 c_y1) (and a!6 (not (= c_x1 c_y1))))))
(let ((a!7 (or a!4
               (not (= (c_f c_x1) c_oct_1))
               (= c_oct_1 (c_f c_x1))
               (and a!6 (not (= c_x1 c_y1))))))
(let ((a!8 (or (= (c_f c_y1) c_oct_2)
               (not (= c_x1 c_y1))
               (and (= (c_f c_x1) c_oct_1) a!7))))
  (and a!8 a!9))))))
Clause Id: 11 (Derived(10,6)) Predicate: false Pivot: (= c_oct_1 c_oct_2)
Pivot is AB-common
Partial interpolant: (let ((a!1 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
                (<= c_oct_1 0)
                (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1))))
      (a!5 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
```

```
               (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2)))
      (a!6 (or (= c_x1 c_y1) a!5 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
(let ((a!4 (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
               (not (= c_oct_1 c_oct_2))))
      (a!9 (or (= c_x1 c_y1) (and a!6 (not (= c_x1 c_y1))))))
(let ((a!7 (or a!4
               (not (= (c_f c_x1) c_oct_1))
               (= c_oct_1 (c_f c_x1))
               (and a!6 (not (= c_x1 c_y1)))))
      (a!10 (or (not (not (= c_oct_1 c_oct_2))) a!4)))
(let ((a!8 (or (= (c_f c_y1) c_oct_2)
               (not (= c_x1 c_y1))
               (and (= (c_f c_x1) c_oct_1) a!7))))
  (and (or (not (= c_oct_1 c_oct_2)) (and a!8 a!9)) a!10))))))
Interpolant((from derived)new): (let ((a!1 (and (>= c_oct_1 0)
               (<= c_oct_1 0)
               (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
               (<= c_oct_1 0)
               (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1))))
      (a!5 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
               (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2)))
      (a!6 (or (= c_x1 c_y1) a!5 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
(let ((a!4 (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
               (not (= c_oct_1 c_oct_2))))
      (a!9 (or (= c_x1 c_y1) (and a!6 (not (= c_x1 c_y1))))))
(let ((a!7 (or a!4
               (not (= (c_f c_x1) c_oct_1))
               (= c_oct_1 (c_f c_x1))
               (and a!6 (not (= c_x1 c_y1))))))
(let ((a!8 (or (= (c_f c_y1) c_oct_2)
               (not (= c_x1 c_y1))
               (and (= (c_f c_x1) c_oct_1) a!7))))
  (and (or (not (= c_oct_1 c_oct_2)) (and a!8 a!9))
       (or (= c_oct_1 c_oct_2) a!4)))))))
Final interpolant for conflict clause: (let ((a!1 (and (>= c_oct_1 0)
               (<= c_oct_1 0)
               (<= (+ (* (- 1) c_oct_2) c_oct_1) (- 1))))
      (a!2 (and (>= c_oct_1 0)
               (<= c_oct_1 0)
               (<= (+ c_oct_2 (* (- 1) c_oct_1)) (- 1))))
      (a!5 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
               (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!3 (and (= 0 c_oct_1) (or (not (= 0 c_oct_1)) a!1 a!2)))
      (a!6 (or (= c_x1 c_y1) a!5 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
(let ((a!4 (and (or (= c_oct_1 c_oct_2) (not (= 0 c_oct_2)) a!3)
               (not (= c_oct_1 c_oct_2))))
      (a!9 (or (= c_x1 c_y1) (and a!6 (not (= c_x1 c_y1))))))
(let ((a!7 (or a!4
               (not (= (c_f c_x1) c_oct_1))
               (= c_oct_1 (c_f c_x1))
               (and a!6 (not (= c_x1 c_y1))))))
(let ((a!8 (or (= (c_f c_y1) c_oct_2)
               (not (= c_x1 c_y1))
               (and (= (c_f c_x1) c_oct_1) a!7))))
  (and (or (not (= c_oct_1 c_oct_2)) (and a!8 a!9))
```

```
       (or (= c_oct_1 c_oct_2) a!4)))))))
-> Final Interpolant: (let ((a!1 (and (<= (+ (* (- 1) c_x1) c_y1) 0)
                (<= (+ c_x1 (* (- 1) c_y1)) (- 1)))))
(let ((a!2 (or (= c_x1 c_y1) a!1 (<= (+ (* (- 1) c_x1) c_y1) (- 1)))))
(let ((a!3 (or (not (= (c_f c_x1) 0))
               (= 0 (c_f c_x1))
               (and a!2 (not (= c_x1 c_y1)))))
      (a!5 (or (= c_x1 c_y1) (and a!2 (not (= c_x1 c_y1))))))
(let ((a!4 (or (not (= c_x1 c_y1)) (= (c_f c_y1) 0) (and (= (c_f c_x1) 0) a!3))))
  (and a!4 a!5)))))
Interpolant:
(let ((a!1 (and (<= (+ (* (- 1) x1) y1) 0) (<= (+ x1 (* (- 1) y1)) (- 1)))))
(let ((a!2 (or (= x1 y1) a!1 (<= (+ (* (- 1) x1) y1) (- 1)))))
(let ((a!3 (or (not (= (f x1) 0)) (= 0 (f x1)) (and a!2 (not (= x1 y1)))))
      (a!5 (or (= x1 y1) (and a!2 (not (= x1 y1))))))
(let ((a!4 (or (not (= x1 y1)) (= (f y1) 0) (and (= (f x1) 0) a!3))))
  (and a!4 a!5)))))
rm -rf tests/*.o tests/basic_test
```

Appendix C: additional implementation code of relevant data structures

6.3 Congruence Closure with Explanation operation header

```
class Hornsat;

class CongruenceClosureExplain : public CongruenceClosure {

  Hornsat * hsat;

  PendingElements pending_elements;
  PendingPointers equations_to_merge;
  PendingPointers pending_to_propagate;

  FactoryCurryNodes const & factory_curry_nodes;

  LookupTable lookup_table;
```

```
14  UseList     use_list;
17
18  void pushPending(PendingPointers &, const PendingElement &);
19  void merge();
20  void merge(EquationCurryNodes const &);
21  void propagate();
22  void propagateAux(CurryNode const &, CurryNode const &,
       EqClass, EqClass, PendingElement const &);
23
24  EqClass         highestNode(EqClass, UnionFind &);
25  EqClass         nearestCommonAncestor(EqClass, EqClass,
       UnionFind &);
26  PendingPointers explain(EqClass, EqClass);
27  void            explainAlongPath(EqClass, EqClass, UnionFind
        &, ExplainEquations &, PendingPointers &);
28  std::ostream &  giveExplanation(std::ostream &, EqClass,
       EqClass);
29
30  public:
31  CongruenceClosureExplain(Hornsat *, CongruenceClosureExplain
        const &, UnionFindExplain &);
32  CongruenceClosureExplain(Z3Subterms const &,
       UnionFindExplain &, FactoryCurryNodes &, IdsToMerge const
        &);
33  ~CongruenceClosureExplain();
34
35  bool areSameClass(EqClass, EqClass);
36  bool areSameClass(z3::expr const &, z3::expr const &);
37
38  EqClass  constantId(EqClass);
```

```
39    EqClass  find(EqClass);
42    z3::expr z3Repr(z3::expr const &);
43
44    void merge(EqClass, EqClass);
45    void merge(z3::expr const &, z3::expr const &);
46
47    PendingPointers explain(z3::expr const &, z3::expr const &);
48    std::ostream &  giveExplanation(std::ostream &, z3::expr
         const &, z3::expr const &);
49
50    z3::expr_vector z3Explain(z3::expr const &, z3::expr const
         &);
51    std::ostream &  z3Explanation(std::ostream &, const z3::expr
          &, const z3::expr &);
52
53    friend std::ostream & operator << (std::ostream &, const
         CongruenceClosureExplain &);
54 };
```

6.4 Hornsat (Gallier's data structure) header

```
1  class Hornsat {
2
3    friend class EUFInterpolant;
4
5    unsigned num_hcs, num_literals;
6    // This structure is only used in our approach
7    // for conditional-elimination
8    std::unordered_map<unsigned, HornClause *> head_term_indexer
        ;
```

```
9
12   UnionFindExplain          ufe;
13   CongruenceClosureExplain equiv_classes;
14
15   std::vector<Literal>    list_of_literals;
16   ClassList               class_list;
17   std::vector<unsigned>   num_args;
18   std::vector<LiteralId> pos_lit_list;
19   // 'facts' is a queue of all the (temporary)
20   // literals that have value true
21   std::queue<LiteralId>   facts;
22   std::queue<TermIdPair> to_combine;
23
24   bool consistent;
25
26   void satisfiable();
27   void closure();
28
29  public:
30   Hornsat(CongruenceClosureExplain &, HornClauses const &);
31   ~Hornsat();
32
33   void build(CongruenceClosureExplain &, HornClauses const &);
34   bool isConsistent() const ;
35   void unionupdate(LiteralId, LiteralId);
36   friend std::ostream & operator << (std::ostream &, Hornsat
        const &);
37 };
```

Bibliography

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[2] Armin Biere. Picosat essentials. *JSAT*, 4:75–97, 05 2008.

[3] Andreas Blass and Yuri Gurevich. Inadequacy of computable loop invariants. *ACM Trans. Comput. Logic*, 2(1):1–11, January 2001.

[4] Maria Paola Bonacina and Moa Johansson. On interpolation in decision procedures. In Kai Brünnler and George Metcalfe, editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 1–16, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[5] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Universitext. Springer Berlin Heidelberg, 2001.

[6] Roberto Bruttomesso, Silvio Ghilardi, and Silvio Ranise. Quantifier-free interpolation in combinations of equality interpolating theories. *ACM Trans. Comput. Logic*, 15(1), March 2014.

[7] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In David L. Dill, editor, *Computer Aided Verification*, pages 68–80, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[8] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. Proof tree preserving interpolation. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 124–138, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[9] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Interpolant generation for utvpi. In Renate A. Schmidt, editor, *Automated Deduction – CADE-22*, pages 167–182, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[10] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, September 2003.

[11] John Cocke. *Programming Languages and Their Compilers: Preliminary Notes*. New York University, USA, 1969.

[12] William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(3):269–285, 1957.

[13] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[14] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[15] Leonardo de Moura and Nikolaj Bjørner. Proofs and refutations and z3, 01 2008.

[16] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *The Journal of Logic Programming*, 1(3):267 – 284, 1984.

[17] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, October 1980.

[18] Herbert B. Enderton. *A mathematical introduction to logic*. Academic Press, 1972.

[19] Alexander Fuchs, Amit Goel, Jim Grundy, Sava Krstić, and Cesare Tinelli. Ground interpolation for the theory of equality. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 413–427, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[20] Bernard A. Galler and Michael J. Fisher. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, May 1964.

[21] Jean H. Gallier. Fast algorithms for testing unsatisfiability of ground horn clauses with equations. *Journal of Symbolic Computation*, 4(2):233 – 254, 1987.

[22] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. *SIGPLAN Not.*, 39(1):232–244, January 2004.

[23] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[24] Deepak Kapur. A new algorithm for computing (strongest) interpolants over quantifier-free theory of equality over uninterpreted symbols. *Manuscript*, 2017.

[25] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley Professional, 1st edition, 2015.

[26] Yuichi Komori. Logics without craig's interpolation property. *Proc. Japan Acad. Ser. A Math. Sci.*, 54(2):46–48, 1978.

[27] Laura Kovács and Andrei Voronkov. Interpolation and symbol elimination. In Renate A. Schmidt, editor, *Automated Deduction – CADE-22*, pages 199–213, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[28] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[29] Shuvendu K. Lahiri and Madanlal Musuvathi. An efficient decision procedure for utvpi constraints. In Bernhard Gramlich, editor, *Frontiers of Combining Systems*, pages 168–183, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[30] K. L. McMillan. Interpolation and sat-based model checking. In Warren A. Hunt and Fabio Somenzi, editors, *Computer Aided Verification*, pages 1–13, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[31] K. L. McMillan. An interpolating theorem prover. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 16–30, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[32] Kenneth McMillan. Interpolants from z3 proofs. In *Formal Methods in Computer-Aided Design*, October 2011.

[33] Elliott Mendelson. *Introduction to Mathematical Logic*. Chapman and Hall-CRC, 5th edition, 2009.

[34] Antoine Miné. The octagon abstract domain. *CoRR*, abs/cs/0703084, 2007.

[35] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 530–535, 2001.

[36] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, October 1979.

[37] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, April 1980.

[38] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:53–58, 2014 (published 2015).

[39] Robert Nieuwenhuis and Albert Oliveras. Congruence closure with integer offsets. In Moshe Y. Vardi and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 78–90, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[40] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In Jürgen Giesl, editor, *Term Rewriting and Applications*, pages 453–468, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[41] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[42] Andrey Rybalchenko and Viorica Sofronie-Stokkermans. Constraint solving for interpolation. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 346–362, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[43] Dirk van Dalen. *Logic and structure (3. ed.)*. Universitext. Springer, 1994.

[44] Georg Weissenbacher. Interpolant strength revisited. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, pages 312–326, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[45] Greta Yorsh and Madanlal Musuvathi. A combination method for generating interpolants. In Robert Nieuwenhuis, editor, *Automated Deduction – CADE-20*, pages 353–368, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.