Approximating and Interpolating Theories of Arithmetic for Software Verification

Author(s): Brillout, Angelo
Publication Date: 2011
Permanent Link: https://doi.org/10.3929/ethz-a-006712356

Rights / License: In Copyright - Non-Commercial Use Permitted
Approximating and Interpolating Theories of Arithmetic for Software Verification

A dissertation submitted to
ETH ZURICH

for the degree of
Doctor of Sciences

presented by
Angelo Brillout
Dipl. Inf. TH Karlsruhe
M. Sc. INSA de Lyon
born December 8, 1980
in Paris / France

accepted on the recommendation of

Prof. Dr. Jürg Gutknecht, examiner
Prof. Dr. Daniel Kröning, co-examiner
Prof. Dr. Armin Biere, co-examiner

2011
Software verification aims at formally proving or disproving the correctness of programs with respect to a formal specification. Yet, existing verification tools typically do not scale to large industrial programs. A pivotal approach to improve scalability is based on the use of approximations, which remove details of the system in order to increase efficiency. This dissertation reports and discusses novel techniques for deriving approximations for programs modeled in theories of arithmetic, namely Presburger and floating-point arithmetic.

A well-established technique to compute approximations is based on Craig interpolation which, for example, can be utilized to obtain approximate postconditions. However, in the case of software modeled with Presburger arithmetic, no efficient interpolation algorithms are known. Our work presents a sound and complete interpolating calculus capable of deriving interpolants for any closed proof. As Presburger arithmetic is not enough to express common program operations, we extend our calculus to uninterpreted predicates and functions. In addition, we identify fragments of the logic which are closed under interpolation. Experimental results show the efficiency of our proposed interpolation procedure compared to the naive and inefficient approach based on quantifier elimination.

Another focus of our work is the computation of abstractions in the presence of floating-point operations which often occur in embedded systems. This thesis shows that classical over- and underapproximation techniques are insufficient to improve efficiency of decision procedures supporting floating-point arithmetic. We propose a general framework for building approximations using a mixture of over- and underapproximations and apply the technique to a sound and complete decision procedure of floating-point arithmetic. Experimental evaluation indicates improved efficiency compared to an abstraction-refinement scheme based on an alternation of over- and underapproximations.


First, I would like to thank Prof. Dr. Daniel Kröning for the opportunity of performing my PhD studies in his research group. I thank Prof. Dr. Jürg Gutknecht for welcoming me in his group after Daniel’s move to Oxford as well as Prof. Dr. Armin Biere for providing feedback on my work. In addition, Prof. Dr. Thomas Gross successfully helped me solving intricate organizational issues arising at the end of my PhD.

I am deeply grateful to Thomas Wahl and Philipp Rümer without whom this thesis would not have been possible. They both took on the role of unofficial co-supervisors by guiding and supporting me from the distance. I will not forget the uncountable hours of discussions on the phone and in Oxford.

Particularly, I thank Bettina Sobottka for her emotional support during my PhD studies. Her confidence and encouragement significantly contributed to the successful realization of this thesis.
# Contents

## 1 Introduction

1.1 State of the Art ................................................. 3  
1.2 Contributions .................................................... 4  
1.3 Outline .......................................................... 6

## 2 Background

2.1 First Order Logic and Theories ................................. 7  
2.1.1 Propositional and First Order Logic ....................... 7  
2.1.2 Theories and Presburger Arithmetic ..................... 10  
2.2 Calculi and Decision Procedures ............................. 13  
2.2.1 Calculi ....................................................... 13  
2.2.2 Decision Procedures ......................................... 15  
2.3 Verification Techniques ........................................ 17  
2.3.1 Approximation and Interpolation Methods ............... 19  
2.3.2 Existing Model Checking Techniques .................... 21

## 3 Interpolation for Presburger Arithmetic

3.1 Motivation ....................................................... 29  
3.2 An Interpolating Sequent Calculus for PA .................. 33  
3.2.1 Propositional, Initialization, and Closure Rules ........ 35  
3.2.2 Rewriting Rules for Equality, Inequality and Divisibility 37  
3.2.3 Soundness and Completeness ............................... 40  
3.3 Mixed Cuts and Strengthening ................................. 42  
3.3.1 Mixed Cuts ................................................... 42  
3.3.2 Successive Strengthening .................................... 43
## CONTENTS

5.3.2 Multiplication and Division ........................................ 76  
5.4 Approximating Floating-Point Arithmetic ............................... 78  
  5.4.1 Overapproximation ................................................. 78  
  5.4.2 Underapproximation ................................................ 82  
5.5 Experimental Results .................................................. 84  
5.6 Related Work ............................................................ 86  
5.7 Summary ................................................................. 87  

6 Mixed Abstractions for FPA .................................................. 89  
  6.1 Motivation .................................................................... 89  
  6.2 The Mixed Abstraction Framework ..................................... 90  
    6.2.1 Overview ............................................................. 90  
    6.2.2 Soundness and Completeness ..................................... 92  
  6.3 Applications to Floating-Point Arithmetic ............................. 93  
    6.3.1 Alternating Abstractions .......................................... 93  
    6.3.2 Genuinely Mixed Abstractions ..................................... 94  
  6.4 Experimental Results ................................................... 96  
    6.4.1 Implementation and Benchmarks ................................. 96  
    6.4.2 Results ............................................................... 97  
  6.5 Related Work ............................................................ 97  
  6.6 Summary ................................................................. 99  

7 Conclusion ........................................................................ 101  
  7.1 Summary of the Contributions .......................................... 101  
  7.2 Future Work .............................................................. 102  

Appendices ................................................................. 104
CONTENTS

A  Rules of the Non-Interpolating Calculus 105

B  Proofs and Auxiliary Lemmas 107
  B.1  Properties of the Interpolating Calculus 107
    B.1.1  Soundness 107
    B.1.2  Completeness 111
    B.1.3  Soundness of the $k$-STRENGTHEN-Rule 113
    B.1.4  Chain interpolation property 116
  B.2  PA+UP is not Closed under Interpolation 127
  B.3  Proofs generating PAID+UP Interpolants 128
    B.3.1  Sufficient Conditions for PAID+UP Interpolants 128
    B.3.2  Completeness of the PAID+UP Calculus 131
  B.4  Integer Projection Lemma 134
  B.5  Proofs generating PAID+UF Interpolants 137
    B.5.1  Sufficient Conditions for PAID+UF$_p$ Interpolants 137
    B.5.2  Closure of PAID+UF under Interpolation 138

C  Detailed Experimental Results 141

Bibliography 151

Curriculum Vitae 153
SOFTWARE affects many aspects of modern life. Ordinary tasks such as starting a car, turning on a microwave oven or making a debit card payment involve software. In addition, programs are essential to control safety-critical systems like nuclear power plants, traffic control and alert systems. In this context, failures of critical software can have devastating consequences. Prominent examples are the Ariane 5 disaster, which was caused by an out-of-bounds floating-point conversion and the postponed opening of Denver’s airport by 9 months due to a software error in the baggage handling system.

Testing is the most widespread technique to detect defects in a software engineering project. The main advantage of testing is its scalability to very complex systems, as it only requires the running a piece of software on a predefined set of inputs, called tests. However, exhaustive testing of all program executions, called traces, is practically infeasible since only a small subset of these traces can be analyzed. In other words, testing can show the presence of errors but not their absence and in the case where no defect was found, correctness is not guaranteed.

A therefore popular approach to improve software reliability is to use formal verification techniques. Given a formal specification, their aim is to prove or disprove the absence of software flaws with mathematical rigor. In contrast to testing, which only explores some of the traces, formal verification conducts an exhaustive exploration of all possible traces. Thus, if a program is declared cor-
rect using a formal verification technique, it implies that the program is error-free with respect to the specification. In addition, a particular verification technique called *model checking* is able to provide faulty traces in the case where a specification is proved to be violated.

Formal methods have received increasing attention by industry and have been applied successfully in the past. For example, Microsoft has been using formal verification to find many flaws in Windows device drivers. Deadlocks have been detected in online reservation systems and major design flaws have been identified in the Deep Space aircraft controller. But despite the growing complexity of software systems, the use of these methods is still languishing. A major reason for this evolution concerns complexity and decidability issues. In general, formal verification of algorithms is undecidable and despite intensive research, formal methods do not scale to large and complex systems.

A common approach to addressing scalability issues is to use *approximations*. Their goal is to simplify details or even remove entire components of the system to increase the efficiency of verification. The simplified system, however, usually does not satisfy the same properties as the original program potentially leading to wrong results during verification. To avoid such wrong outcomes, spuriousness needs to be detected and approximations need to be refined to eventually reach a sound output.

Approximations can mostly be classified in two categories. Overapproximations enrich the behavior of programs by relaxing constraints. In this case, if no error is found the original system is error-free too, while a violating execution path might be spurious. Conversely, underapproximations remove behavior from the system so that a violating trace in the abstracted system corresponds to a bug in the original program.

This thesis provides novel techniques to derive approximations for programs modeled using theories of arithmetic, namely Presburger and floating-point arithmetic.

- We provide new over- and underapproximation methods for programs containing floating-point operations. To improve the accuracy of the approximations, we propose a technique to combine both over- and underapproximations to obtain so called *mixed abstractions*.
- We present a Craig interpolation procedure for software modeled in Pres-
1.1 State of the Art

A frequently used theory to model the behavior of infinite-state programs is Presburger arithmetic, i.e., the first order theory of linear integer arithmetic. The decidability of the theory has led to numerous implementations of complete decision procedures (e.g. [66, 49, 32, 73]) and, since many programs manipulate integers, Presburger arithmetic is a natural choice to encode software. But despite its popularity, no efficient and complete interpolation algorithms exist for this theory. Interpolation procedures have so far been reported only for restricted fragments of Presburger arithmetic [58, 76, 59, 17, 44]. Consequently, interpolation techniques are not applicable to systems modeled with the full theory of Presburger arithmetic.

To faithfully model software, full Presburger arithmetic is not sufficient. Programming notions, such as arrays, structures and variable multiplication do not have any equivalent in pure Presburger arithmetic. Moreover, machine addition does not correspond to mathematical addition. Consequently, the theory needs to be extended to express the behavior of such operations and data structures. Yet, any theory extension also requires an extension of its interpolation procedure. Even though many approaches have been proposed to interpolate in the presence of uninterpreted function or array-like data structures (e.g. [81, 58, 35]), no interpolation procedure is capable of handling the full theory of Presburger arithmetic with arrays.

Another challenge to find flaws in software is the lack of support for floating-point operations. Although they are commonly used in programming languages such as C or Java, no complete decision procedure supporting floating-point operations is known. Incomplete procedures based on abstract interpretation and
proof assistants exist (e.g. [5, 55, 67]), but no decision procedure is able to
disprove correctness in the presence of floating-point operations. As shown in
this thesis, however, such operations are very complex to model accurately and
neither overapproximations nor underapproximations are sufficient to increase
efficiency. Thus, new approximation techniques are required to render the verific-
atation feasible in practice. Moreover, to be complete, new approximation methods
must be able to refine approximations in case spurious results are detected.

This work proposes new approximation techniques to tackle the afore men-
tioned shortcomings. We summarize the results of our work in the following
section.

1.2 Contributions

Our contributions can be summarized as follows.

This thesis presents new methods to derive approximations for pro-
grams modeled in theories of arithmetic, namely Presburger and
floating-point arithmetic. One aspect of this work pertains to inter-
polation which is a well-known technique to compute approxim-
ations.

In more detail, this thesis discusses and addresses the following aspects:

**Interpolation technique for Presburger arithmetic.** We introduce a novel in-
terpolating sequent calculus for quantifier-free Presburger arithmetic resulting
in the first interpolation method capable of deriving interpolants in this theory
(besides the brute-force technique that is quantifier-elimination). The method is
based on a sequent calculus whose proof rules are annotated to yield interpolants
at the root of a validity proof. We demonstrate the correctness and completeness
of the calculus and we present several practical optimizations. The proposed
calculus is implemented on top of a theorem prover based on the Omega-test.
Experimental results show the robustness and efficiency of the procedure.
1.2. Contributions

Interpolation for extensions of Presburger arithmetic. To use interpolation techniques for the verification of programs, the interpolating sequent calculus for Presburger arithmetic is extended to uninterpreted predicates and functions. By means of a relational encoding of functions and array operations, the interpolation procedure is extended to support uninterpreted functions and the theory of arrays. However, we prove that in this theory some formulae do not admit quantifier-free interpolants. We further extend the calculus to handle quantifiers and we identify fragments with restricted forms of quantification that are closed under interpolation. Owing to these extensions, we apply the interpolating calculus to the verification of software using the lazy abstraction framework. To ensure soundness, we prove that the calculus satisfies the “chain interpolation” property which is required by the framework. For the verification of C-programs, we also provide encodings of common C-operations such as classical bit-vector operations.

Approximation techniques for floating-point arithmetic. We present new under- and overapproximation techniques to significantly simplify formulae containing floating-point operations. The approximation is achieved by reducing the precision of these operations as well as by modifying rounding decisions. Using these approximations, we present a sound and complete decision procedure for floating-point arithmetic based on the alternating abstraction framework. A distinguishing feature of the proposed procedure is to produce witnesses in the case where a property is found to be violated. Experimental results suggest that, if rounding is crucial to the outcome of the verification, strict over- or underapproximations are not sufficient to significantly increase the efficiency of verification. We address this using the mixed abstraction framework presented next.

Mixed abstraction for floating-point arithmetic. A new approach that mixes both over- and underapproximations to obtain more precise approximations is proposed. The technique proceeds by applying predefined over- and underapproximation transformations to a formula containing floating-point operations. The resulting approximation, which is neither an under- nor an overapproximation, is passed to a decision procedure capable of generating witnesses in both unsatisfiability and satisfiability cases. Using these witnesses, our method detects the transformations leading to spurious results which are removed to generate a
refined approximation. Recursively calling the procedure and making sure that the set of transformation eventually depletes, yields a complete decision procedure for theory of floating-point arithmetic. We implemented the proposed method in a bounded model checker capable of handling floating-point operations in a bit-precise manner. Experimental results show the potency of the procedure for the analysis of floating-point software.

1.3 Outline

This dissertation is organized as follows:

- The background material such as first order logic and theories, calculi and decision procedures as well as common abstraction and verification techniques are explained in Chapter 2.

- Chapter 3 presents an interpolating calculus for the full theory of Presburger arithmetic. Several practical optimizations and an experimental evaluation of the proposed procedure are given.

- Chapter 4 describes an extension of the interpolation procedure to support uninterpreted predicates and functions. In addition, fragments of these extensions which are closed under interpolation are identified.

- Chapter 5 proposes novel over- and underapproximation techniques for floating-point arithmetic. The techniques are implemented and evaluated in the existing framework of [12].

- Chapter 6 shows how to construct more accurate approximations for floating-point arithmetic by mixing over- and underapproximations. We give a refinement procedure that subsumes the framework of [12] and experimentally evaluate the proposed technique.

- We conclude and discuss future directions of this work in Chapter 7.
2.1 First Order Logic and Theories

2.1.1 Propositional and First Order Logic

**Propositional Logic.** We begin the definition of propositional logic with a description of its syntax which determines the legal expressions in the logic. This is followed by the definition of the semantics, i.e., the meaning of these expressions.

Let $p$ range over a finite set $P$ of atomic propositional symbols, also called (0-ary) predicates. The set of propositional formulae $L_{PL}$ over $P$ is defined by the grammar

$$
\phi ::= (\phi \land \phi) \mid (\phi \lor \phi) \mid \neg \phi \mid p \mid true \mid false
$$

(2.1)

where the symbol $\neg$ is the negation, $\land$ the conjunction and $\lor$ the disjunction operator. The implication operator $\phi \rightarrow \psi$ is a shorthand for $\neg \phi \lor \psi$ and the equivalence operator $\phi \leftrightarrow \psi$ stands for $(\phi \rightarrow \psi) \land (\psi \rightarrow \phi)$. A predicate or its negation is called literal. A disjunction of literals is called clause. A propositional formula is said to be in conjunctive normal form (CNF) if it is a conjunction of clauses.

The semantics of propositional formulae is defined via a valuation function. Let $\mathbb{B} := \{False, True\}$ be the set of Boolean values and $I: P \rightarrow \mathbb{B}$ be an assignment of truth values to predicates (also called interpretation function). A propositional
valuation is a function $\text{val}_I : \mathcal{L}_{PL} \to \mathbb{B}$ recursively defined as

\[
\begin{align*}
\text{val}_I(\text{true}) &:= \text{True} \\
\text{val}_I(\text{false}) &:= \text{False} \\
\text{val}_I(p) &:= I(p) \text{ for a 0-ary } p \in P \\
\text{val}_I(\phi \land \psi) &:= \text{True} \text{ if } \text{val}_I(\phi) = \text{True} \text{ and } \text{val}_I(\psi) = \text{True} \\
\text{val}_I(\phi \lor \psi) &:= \text{True} \text{ if } \text{val}_I(\phi) = \text{True} \text{ or } \text{val}_I(\psi) = \text{True} \\
\text{val}_I(\neg \phi) &:= \text{True} \text{ if } \text{val}_I(\phi) = \text{False} \\
&:= \text{False} \text{ otherwise}
\end{align*}
\]

The interpretation function $I$ can be seen as an “environment” that declares which predicates hold and which do not hold. The $\text{val}_I$ function then recursively evaluates the truth value of an arbitrary formula $\phi$ based on the interpretation $I$. To illustrate this, consider the following example.

**Example 2.1.1.** Given the formula $p \land \neg q$ and let $P := \{p, q\}$, $I(p) := \text{True}$ and $I(q) := \text{False}$. The formula is evaluated as follows:

\[
\text{val}_I(p \land \neg q) = \text{True} \quad \text{if} \quad \text{val}_I(p) = \text{True} \text{ and } \text{val}_I(\neg q) = \text{True} \\
\text{since} \quad \text{val}_I(p) = \text{True} \text{ and } \text{val}_I(q) = \text{False}
\]

**First order logic.** First order logic (FOL) is an extension of propositional logic that increases the expressiveness of the language. It permits n-ary predicates as well as quantifiers in order to express properties of objects which are represented by variables. First order logic, for example, is capable of expressing facts such as “all elements $a$ of an infinite set satisfy the unary predicate $p(a)$”. Indeed, such an expression does not have any equivalent formulae in propositional logic.

The set of first order formulae denoted $\mathcal{L}_{FOL}$ is simply defined by adding the following rule to the grammar (2.1) of propositional logic:

\[
\phi ::= \forall x.\phi \mid \exists x.\phi \mid p(x, \cdots, x) \tag{2.2}
\]

in which $x$ ranges over a set of variables $V$ and $p$ over a set of predicate symbols $P$ with arbitrary arity. The symbols $\forall$ and $\exists$ are called quantifiers and, for a given formula, a variable is said to be bound if it is in the scope of a quantifier and to
be free otherwise. In the following, let $\mathcal{P}(S)$ denote the power set and $S^n$ be the set of all possible $n$-tuples of a set $S$.

The semantics of FOL is easily defined by extending the valuation function $val_I$ to be defined on quantifiers and $n$-ary predicates. To this end, let $U$ be a non-empty set called universe and let $I$ be an interpretation function also defined on $n$-ary predicates and variables. More precisely, the function $I$ maps truth values in $B$ to 0-ary predicate symbols in $P$, $n$-ary relations in $\mathcal{P}(U^n)$ to $n$-ary predicate symbols in $P$ and values in $U$ to variables in $V$. We define an extended valuation operator $val_{U,I} : \mathcal{L}_{FOL} \rightarrow B$ as follows. On formulae with outermost symbol $\land, \lor, \neg$ or a 0-ary predicate symbol, the operator $val_I$ for propositional logic. On the non-propositional symbols, the valuation operator is defined as:

$$
val_{U,I}(p(x_1, \cdots, x_n)) := True \text{ if } (I(x_1), \cdots, I(x_n)) \in I(p) \text{ for an } n\text{-ary } p \in P
$$

$$
val_{U,I}(\forall x. \phi) := True \text{ if for all } o \in U : val_{U,I^o_x}(\phi) = True
$$

$$
val_{U,I}(\exists x. \phi) := True \text{ if there is } o \in U : val_{U,I^o_x}(\phi) = True
$$

$$
:= False \text{ otherwise}
$$

where $I^o_x$ denotes the interpretation function identical to $I$ except that $I^o_x(x) := o$ for a variable $x \in V$ and an object $o \in U$. We illustrate the valuation of formulae with the following example.

**Example 2.1.2.** Consider the formula $\forall x. \exists y. p(x,y)$ and let $U := \{a,b\}$ and $I(p) := \{(a,b),(b,a)\}$. The formula is evaluated as follows.

$$
val_{U,I}(\forall x. \exists y. p(x,y)) = True \text{ if } \text{for all } o \in U \text{ val}_{U,I^o_x}(\exists y. p(x,y)) = True
$$

$$
\text{ if } \text{for all } o \in U \text{ there exists } o' \in U \text{ such that } val_{U,I^o_x^o'}(p(x,y)) = True
$$

$$
\text{ if } \text{for all } o \in U \text{ there exists } o' \in U \text{ such that } val_{U,I}(p(o,o')) = True
$$

$$
val_{U,I}(\forall x. \exists y. p(x,y)) = True \text{ since } (a,b) \in I(p) \text{ and } (b,a) \in I(p)
$$
Further Terminology. For first order logic, the pair \((U, I)\) is called a *structure* and for a given formula \(\phi\), a structure is said to be a *satisfying structure* \(^1\) for \(\phi\) if \(val_{(U, I)}(\phi) = \text{True}\). For propositional logic, a structure reduces to the assignment \(I\) since no universe is required for the evaluation of propositional formulae. Thus, a satisfying structure is also called a *satisfying assignment* in propositional logic. A formula is said to be *satisfiable* if it has a satisfying structure and *unsatisfiable* otherwise. If all structures satisfy \(\phi\), we say that \(\phi\) is *valid*.

The symbols \(\wedge, \vee, \neg, \text{true}, \text{false}, \exists\) and \(\forall\) are called *logical symbols* since they have a predefined meaning in propositional and first order logic. On the other hand, the predicate symbols in \(P\) are called *non-logical* or *uninterpreted* symbols since their meaning is defined externally through an interpretation function \(I\).

### 2.1.2 Theories and Presburger Arithmetic

A *theory* \(\mathcal{T}\) is a set of satisfiable formulae which is closed under logical deduction. That is, for any finite subset \(\Phi \subseteq \mathcal{T}\) and any formula \(\psi\) such that \(\wedge \Phi \rightarrow \psi\) is valid it follows that \(\psi \in \mathcal{T}\). The formulae in theory \(\mathcal{T}\) are called *theorems*. The *axiomatic* approach to defining theories is to give a decidable set of satisfiable formulae \(\mathcal{A}\), called *axioms*, and to define the corresponding theory as the closure of \(\mathcal{A}\) under logical deduction:

\[
\text{Ded}(\mathcal{A}) := \{ \psi \mid \text{there exists a subset } \Phi \subseteq \mathcal{A} \text{ such that } \wedge \Phi \rightarrow \psi \text{ is valid} \}
\]

Equivalently, this means that a formula \(\psi\) belongs to \(\text{Ded}(\mathcal{A})\) iff it is satisfied by all the structures \(\mathcal{S}\) satisfying the axioms. The *model theoretical* approach to defining theories, is to directly provide a set of structures \(\mathcal{S}\) and to define its corresponding theory as the set of formulae satisfied by all structure in \(\mathcal{S}\):

\[
\text{Th}(\mathcal{S}) := \{ \psi \mid \text{all structures in } \mathcal{S} \text{ satisfy } \psi \}
\]

In the context of theories, the evaluation of a formula is restricted to structure in \(\mathcal{S}\) and the notion of validity, satisfiability and unsatisfiability are adapted accordingly. That is, a formulae is said to be *satisfiable* (in a theory \(\mathcal{T}\)) if it is satisfied by some structures in \(\mathcal{S}\) and unsatisfiable otherwise. A formula that is satisfied by all structure \(\mathcal{S}\) is said to be valid (in \(\mathcal{T}\)).

\(^1\)In mathematical logic, a satisfying structure is also called a *model*. Yet, in this thesis, we use the term model to denote a (formal) representation of a system.
Next, we describe the common theories of equality and uninterpreted functions, as well as Presburger arithmetic, a theory used throughout this work.

Equality theory. One of the most important theories in practice is the theory of equality. For an arbitrary universe $U$ and a set of predicate symbols $P$ containing the binary symbol $\equiv$, the theory of equality can be defined using the following axioms.

\[
\forall x. x \equiv x \quad \text{(reflexivity)}
\]

\[
\forall x_1, y_1, \ldots, x_n, y_n. (x_1 \equiv y_1 \land \cdots \land x_n \equiv y_n) \rightarrow (p(x_1, \cdots, x_n) \iff p(y_1, \cdots, y_n)) \quad \text{(monotonicity)}
\]

Essentially the axioms define the notion of equality using the new symbol $\equiv$ and two first order formulae describing reflexivity and monotonicity. An equivalent model theoretical definition is given by the set of all structures, denoted $\langle \equiv \rangle$, in which the interpretation function is restricted to $I(\equiv) := \{(o, o) \mid o \in U\}$. The equality theory obtained is then $Th(\langle \equiv \rangle)$.

Function theory. Functions can be represented with the help of special predicates. More precisely, an $n$-ary functions $f$ is expressible via an $n + 1$-ary predicate $p_f$ satisfying the properties of functionality and totality. That is, the predicate symbol $p_f$ has to satisfy the two axioms:

\[
\forall x_1, \cdots, x_n. \exists y. p_f(x_1, \cdots, x_n, y) \quad \text{(totality)}
\]

\[
\forall x_1, \cdots, x_n, y, y'. p_f(x_1, \cdots, x_n, y) \land p_f(x_1, \cdots, x_n, y') \rightarrow y \equiv y' \quad \text{(functionality)}
\]

where $y$ and $y'$ represent the result of the function.

To simplify the language and the readability of first order formulae, it is common to integrate function symbols into the syntax of first order logic. Let $f$ range over a set of function symbols $F$. The syntax is extended by adding the following rules to the grammar defined by (2.1) and (2.2):

\[
t ::= x \mid f(t, \cdots, t)
\]

\[
\phi ::= p(t, \cdots, t)
\]  

(2.3)
where $x$ ranges over the set of variables $X$. In this rule $t$ is called a *term* and represents applications or compositions of functions.

To define the semantics of functions and terms, the interpretation function $I$ is extended to map $n$-ary functions in $U^n \to U$ to $n$-ary function symbols in $F$. The range of the valuation function is extended to the universe $U$, i.e., the function $\text{val}_{U,I} : \mathcal{L}_{FOL} \to B \cup U$ is defined as before and is extended to be defined on function and terms as

$$\text{val}_{U,I}(x) := I(x) \text{ and } x \in V$$
$$\text{val}_{U,I}(f(t_1, \ldots, t_n)) := I(f)(\text{val}_{U,I}(t_1), \ldots, \text{val}_{U,I}(t_n)) \text{ and } f \in F$$
$$\text{val}_{U,I}(p(t_1, \ldots, t_n)) := \text{True} \text{ if } (\text{val}_{U,I}(t_1), \ldots, \text{val}_{U,I}(t_n)) \in I(p) \text{ and } p \in P$$
$$\text{val}_{U,I}(p(t_1, \ldots, t_n)) := \text{False} \text{ otherwise}.$$

In this thesis, free variables are considered to be interpreted as 0-ary functions. We refer to [34] for more information on propositional and first order logic.

**Presburger arithmetic and extensions**  
Quantifier-free Presburger arithmetic, denoted PA, is the first-order theory of linear *integer* arithmetic with addition. More precisely, it is defined by the set of structures, denoted $\langle \mathbb{Z}, +, \leq, 2 |, 3 |, \ldots \rangle$, in which the universe is the set of integers $\mathbb{Z}$, the only function symbol $+$ is interpreted as regular mathematical addition, the predicate symbol $\leq$ as mathematical inequality and, the predicate symbols $2 |, 3 |, \cdots$ as the *constant divisibility* predicates, i.e.,

$$\text{val}_{\mathbb{Z},I}(2 | t) := \text{True} \text{ if there exists } a \in \mathbb{Z} \text{ with } 2 \cdot a = \text{val}_{\mathbb{Z},I}(t)$$
$$\text{val}_{\mathbb{Z},I}(3 | t) := \text{True} \text{ if there exists } a \in \mathbb{Z} \text{ with } 3 \cdot a = \text{val}_{\mathbb{Z},I}(t)$$

$$\vdots$$

The theory of PA is then defined by $Th(\langle \mathbb{Z}, +, 2 |, 3 |, \cdots \rangle)$. An axiomatization of the theory is given in [34].

Quantified Presburger arithmetic (QPA) is an extension of PA in which quantifiers are allowed in formulae. Further, the theory denoted QPA+UP (resp. QPA+UF) is an extension of QPA in which uninterpreted predicates (resp. functions) are permitted.
2.2 Calculi and Decision Procedures

2.2.1 Calculi

In the context of mathematical logic, a calculus consists of a formal language (such as $\mathcal{L}_{PL}$ and $\mathcal{L}_{FOL}$) and a set of inference rules which is used to derive new conclusions from a given set of premises. These rules are then applied to a formulae or its derived conclusions until validity/unsatisfiability is established. This section briefly presents the resolution calculus and the sequent calculi family which are used in this work.

In the following, we write $\bigwedge \Gamma$ (resp. $\bigvee \Gamma$) to denote the conjunction (resp. disjunction) of all elements in a set of formulae $\Gamma$ and we define $\bigwedge \emptyset := true$ (resp. $\bigvee \emptyset := false$). We use the comma “,” as shorthand for the union operation $\cup$ on sets and we write $\Gamma, e$ to denote $\Gamma \cup \{e\}$.

Resolution Calculus The resolution calculus is a refutation calculus for propositional formulae in CNF consisting of a single rule, namely the resolution rule [70]. Given two sets of literals $\Lambda_1, \Lambda_2$ standing for the clauses $\bigvee \Lambda_1, \bigvee \Lambda_2$ and a predicate symbol $p$, the resolution rule is a binary relation between two clauses called the premises and a clause called the conclusion or resolvent:

$$\frac{\Lambda_1, p \quad \Lambda_2, \neg p}{\Lambda_1, \Lambda_2} \text{ RES}$$

The resolution rule is such that the satisfiability of the conjunction of the clauses in the premises $(\bigvee \Lambda_1 \lor p) \land (\bigvee \Lambda_2 \lor \neg p)$ implies the satisfiability of the clause $\bigvee \Lambda_1, \Lambda_2$ in the conclusion. By contraposition, this means that if the empty clause $\bigvee \emptyset = false$ is derived then the conjunction of the clauses in the premises is unsatisfiable. A resolution proof is a binary tree growing downward. Each node is labeled with a clause and the leaf nodes correspond to clauses in the CNF formulae to be refuted. Each non-leaf node is related to the nodes directly above it through an instance of the resolution rule. The root at the bottom of the tree is labeled with the empty clause.

Sequent Calculi. A widespread family of calculi are called (Gentzen-style) sequent calculi which are based on those first introduced by Gerhard Gentzen in
1934 [37]. If \( \Gamma, \Delta \) are two finite sets of formulae without free variables, then \( \Gamma \vdash \Delta \) is called a *sequent* and \( \Gamma \) (resp. \( \Delta \)) the antecedent (resp. succedent) of the sequent. A sequent is said to be *valid* if the formula \( \land \Gamma \rightarrow \lor \Delta \) is valid. A calculus *rule* is a binary relation between a finite set of sequents called the *premises*, and a sequent called the *conclusion*. A sequent calculus rule is *sound* if, for all instances

\[
\frac{\Gamma_1 \vdash \Delta_1 \quad \cdots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}
\]

whose premises \( \Gamma_1 \vdash \Delta_1, \ldots, \Gamma_n \vdash \Delta_n \) are valid, the conclusion \( \Gamma \vdash \Delta \) is valid, too. A sequent calculus is a set of sequent rules. Proof trees are defined to grow upwards. Each node is labeled with a sequent, and each non-leaf node is related to the node(s) directly above it through an instance of a calculus rule. A proof is *closed* if it is finite and branches are closed, i.e., all leaves are justified by an instance of a rule without premises.

**Example 2.2.1.** The following presents three examples of sequent rules, each of which is given a name.

\[
\frac{\Gamma, \phi \vdash \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \phi \lor \psi \vdash \Delta} \quad \text{OR-LEFT'}
\]

\[
\frac{\Gamma \vdash \phi, \Delta}{\Gamma, \neg \phi \vdash \Delta} \quad \text{NOT-LEFT'}
\]

\[
\frac{*}{\Gamma, \phi \vdash \phi, \Delta} \quad \text{CLOSE'}
\]

The rule called OR-LEFT' splits the proof in two branches, whereas the CLOSE' rule closes a branch, since any sequent of the form \( \Gamma, \phi \vdash \phi, \Delta \) is trivially valid.

Next, we give an example of a closed sequent proof. The root of the proof is the sequent \( p, \neg p \lor q \vdash q \), where \( p, q \) are 0-ary predicates.

\[
\frac{*}{p \vdash p, q} \quad \text{CLOSE'}
\]

\[
\frac{p, \neg p \vdash q}{p, q \vdash q} \quad \text{NOT-LEFT'}
\]

\[
\frac{p, q \vdash q}{p, \neg p \lor q \vdash q} \quad \text{CLOSE'}
\]

By applying the OR-LEFT' rule to the root, the proof is split into two branches, yielding two premises. The NOT-LEFT' rule can be applied to the left premiss in order to close the branch with the CLOSE' rule. The right branch on the other hand can be directly closed with the same rule. This shows the validity of the sequent \( p, \neg p \lor q \vdash q \).
2.2. Decision Procedures

A decision problem is a question in some formal system with a yes/no answer. For example, the problem of whether a given first order formula is satisfiable is a decision problem. Algorithms that solve decision problems are called decision procedures. In the context of formal verification, decision procedures play a central role since verifying if a program satisfies a property is essentially a yes/no question in a formal language.

In the following, we briefly present two well-known decision procedures that decide the satisfiability problem for propositional logic and Presburger arithmetic. Both procedures are able (or can be extended) to return a satisfying structure in case satisfiability is detected. If the unsatisfiability of $\bigwedge \Theta$ is established for a set of formulae $\Theta$, both techniques are also able to provide a proof of unsatisfiability in the form of a resolution proof in the case of propositional logic or a sequent proof of $\Theta \vdash \emptyset$ in the case of Presburger arithmetic. In addition, by analysing the proofs, it is easy to extract an unsatisfiable core, i.e., a subset $\Theta' \subseteq \Theta$ such that $\bigwedge \Theta'$ is still unsatisfiable.

**SAT solvers and DPLL.** The term SAT solver connotes a class of algorithms and tools deciding the satisfiability problem of propositional formulae. Currently most efficient SAT-solvers are variations of the well-known Davis-Putnam-Logemann-Loveland (DPLL, [28]) procedure, which decides the satisfiability of propositional formulae in conjunctive normal form (CNF). In its simplest form, the procedure progresses by choosing a predicate, assigning a truth value to it, simplifying the formula and recursively checking if the simplified formula is satisfiable. In the case of a conflict, the algorithm backtracks and uses the opposite truth value to resume the search.

Upon a decision on the truth value of a predicate, the propositional formula is simplified by eliminating all clauses in which a corresponding literal becomes true and by omitting all literals that are false. For instance, if true is assigned to $x_1$, a clause such as $(x_1 \lor x_2 \lor \neg x_3)$ is eliminated completely since it is satisfied. Similarly, a clause like $(\neg x_1 \lor \neg x_2 \lor x_3)$ is simplified to $(\neg x_2 \lor x_3)$ since $\neg x_1$ does not affect the truth value of the clause if true is assigned to $x_1$.

In addition, the procedure is enhanced to include the following optimizations. If a clause contains a single unassigned literal, it can only be satisfied by assigning
True to the remaining literal. This is known as the \textit{unit clause} rule. Another optimization is to detect predicates that occur with only one polarity (negated or not) in the formula. In that case, a truth value can be assigned to the predicate in a way that satisfies all clauses containing that predicate. Hence, these clauses can be removed from the formula.

\textbf{Omega Test.} The Omega test is an algorithm to decide the satisfiability of quantified Presburger arithmetic (QPA) \cite{66}. The main aspect of the algorithm is the adaptation of the \textit{Fourier-Motzkin} (FM) variable elimination method for inequalities over the reals to the domain of the integers.

Consider a set of inequalities $\Theta$ containing two bounds on a variable $x$ of the form $ax + t \leq 0$ and $-bx + s \leq 0$, where $t, s$ are two terms not containing $x$ and where $a, b$ are two positive integers. The FM variable elimination rule consists of multiplying the first bound by $b$ and the second by $a$ in order to derive the linear combination $bt + as \leq 0$. This is expressed by the following valid sequent rule:

$$
\frac{\Gamma, ax + t \leq 0, -bx + s \leq 0, bt + as \leq 0 \vdash \Delta}{\Gamma, ax + t \leq 0, -bx + s \leq 0 \vdash \Delta}^{\text{FM-ELIM}'}
$$

In real arithmetic, the derived inequality $bt + as \leq 0$ is such that

$$
\exists x. (ax + t \leq 0 \land -bx + s \leq 0) \iff bt + as \leq 0 \quad (2.4)
$$

is valid. In particular, Equation (2.4) entails that the satisfiability of $bt + as \leq 0$ implies the satisfiability of $ax + t \leq 0 \land -bx + s \leq 0$ (and vice versa). On the domain of integers, however, this no longer holds in general. For instance, the satisfiability of $0 \leq 0$ does not imply the satisfiability of $2x - 1 \leq 0 \land -2x + 1 \leq 0$ since $1/2$ is not an integer solution. This means, however, that if no contradiction of the form $c \leq 0$ for a positive integer $c$ can be derived by applying the FM-ELIM' rule on all possible pairs of lower and upper bounds in $\Theta$, then nothing can be concluded on the satisfiability of the formula $\bigwedge \Theta$ (i.e. the validity of $\Theta \vdash \emptyset$). In other words, the FM-ELIM' rule alone does not lead to a \textit{complete} decision procedure for conjunctions of inequalities.

To obtain a complete decision procedure for QPA, W. Pugh introduced \textit{splinters} which are equalities conjoined to the upper and lower bounds in order to restore the validity of Equation (2.4). Given the bounds $ax + t \leq 0$ and $-bx + s \leq 0$, a
2.3. Verification Techniques

splinter \( ax + t + i = 0 \) is added to the conjunction of the bounds as follows:

\[
\exists x. \ (ax + t \leq 0 \land -bx + s \leq 0)
\]

\[ \leftrightarrow \]

\[
bt + as + (a - 1)(b - 1) \leq 0 \lor \bigvee_{0 \leq i \leq m} \exists x. \ (ax + t \leq 0 \land -bx + s \leq 0) \land (ax + t + i = 0)
\]

where \( m := \lfloor (ab - a - b)/b \rfloor \). In geometrical terms, the left disjunct \( bt + as + (a - 1)(b - 1) \leq 0 \) describes the “thick part” wider than 1 of the polyhedron \( ax + t \leq 0 \land -bx + s \leq 0 \). That is, the satisfiability \( bt + as + (a - 1)(b - 1) \leq 0 \) guarantees the existence of an integer solution satisfying \( ax + t \leq 0 \land -bx + s \leq 0 \). The right disjuncts describe the “thin part” of the polyhedron for which any potential integer solution must be on one of the planes described by the splinters. A solution satisfying one of the \( m + 1 \) disjuncts guarantees the existence of a solution satisfying the conjunction of the bounds. This is expressed by the valid \( \text{OMEGA-ELIM}' \) rule that generates \( m + 2 \) premises:

\[
\begin{align*}
\{ \Gamma, ax + t \leq 0, -bx + s \leq 0, ax + t + i = 0 \vdash \Delta \} &_{0 \leq i \leq m} \\
\Gamma, ax + t \leq 0, -bx + s \leq 0, bt + as + (a - 1)(b - 1) \leq 0 & \vdash \Delta \\
\end{align*}
\]

Using the \( \text{OMEGA-ELIM}' \) together with standard rules handling equalities and propositional operators (as presented in Figure A.1 or in [66]) leads the complete decision procedure for QPA known as the Omega Test. We refer the reader to [51] for a more detailed description of the procedure.

2.3 Verification Techniques

Given a formal description of a system as well as a formal specification, model checking is the decision problem of automatically verifying if the system satisfies the specification. In the case where the specification is found to be violated, model checkers are able to provide a counterexample, i.e., an execution path exhibiting the violation. As it is often the case in formal verification, model checking can be reduced to a satisfiability or validity problem in a mathematical theory.
Model checking was independently pioneered by Clarke and Emerson [19] and Queille and Sifakis [68]. At that time, model checking was confined to the verification of finite-state systems by a reduction to a decision problem in temporal logics (usually LTL or CTL [24, 2]). In this context, the finite-state system is modeled using a given Kripke structure (a specific type of structures for temporal logics) and the desired specification is expressed by a temporal formula. Model checking then amounts to checking if the Kripke structure satisfies the temporal formula and returning a counterexample if the formula is violated.

Early model checking algorithms used explicit representations of Kripke structures, enumerating all possible states and transitions of the system. Yet, the exponential growth of the state space, known as the state explosion problem, renders this approach infeasible in practice. To counter this representation problem, McMillan [57] proposed to symbolically represent finite state systems (i.e. the Kripke structure) using binary decision diagrams (BDDs), a canonical form for propositional logic [10]. Such canonical representations, however, also suffer from a worst-case exponential space blow-up [62] which occurs for the representation of machine multiplication, for example [11]. Despite these limitations, BDD-based model checking enabled the verification of systems with state spaces ten orders or magnitude larger than those that could be handled with an explicit representations [14].

An alternative to canonical representations is to use logical formulae as a formal language to describe a specification and the behavior of a system. For example, SAT-based model checking techniques [18] use propositional formulae to concisely describe both the transition relation and the specification of a finite state-system. In this way, efficient satisfiability or validity solvers (like propositional SAT solvers) can be employed to decide the verification problem (e.g. [4, 20]). The bottleneck in this approach, of course, is the satisfiability or validity check of the formula (in the case of propositional formulae the problem is NP-complete, i.e., it is likely to take deterministically exponential time in the worst case).

Over- and underapproximations techniques are popular techniques to improve the scalability of automated formal verification methods. In general, their goal is to simplify the system under verification in order to curb complexity and decidability issues. For infinite-state systems, for example, finite-state approximations can be sufficient to verify certain properties using finite-state model checking, although software verification is undecidable in general [69]. Next, we define the
2.3. Verification Techniques

notions of over- and underapproximations and briefly describe existing model checking techniques relying on approximations.

2.3.1 Approximation and Interpolation Methods

Approximation\(^2\) is considered to be one of the most important approaches to addressing scalability issues of formal verification. In the context of systems modeled using mathematical formulae, applying approximation techniques amounts to simplifying formulae so as to improve the efficiency of decision procedures checking for satisfiability or validity. In most cases, formulae are *approximated* in one of two ways:

**Overapproximation.** A formula \(\overline{\phi}\) is said to be an *overapproximation* of a formula \(\phi\) iff the implication \(\phi \rightarrow \overline{\phi}\) is valid. Intuitively, an overapproximation permits *more* satisfying assignments. This means that if a decision procedure reports unsatisfiability for an overapproximation, we can conclude that \(\phi\) is unsatisfiable too. In the case of satisfiability, however, nothing can be concluded about the satisfiability of \(\phi\) since a satisfying structure for \(\overline{\phi}\) (i.e. the model) may be spurious for \(\phi\). In the light of formal verification, if \(\phi\) encodes violating execution traces of a program, then \(\overline{\phi}\) represents *more* faulty traces than \(\phi\). The unsatisfiability of \(\overline{\phi}\) implies that the safety property holds for the original program, while a satisfying assignment (i.e. violating trace) might be spurious. As a result, overapproximations can be used to prove correctness, viz., the absence of a violating trace.

**Underapproximation.** A formula \(\phi\) is said to be an *underapproximation* of a formula \(\overline{\phi}\) iff \(\overline{\phi} \rightarrow \phi\) is valid. As opposed to overapproximations, underapproximations permit *fewer* satisfying assignments. This means that the satisfiability of an underapproximation \(\overline{\phi}\) implies the satisfiability of \(\phi\), while unsatisfiability is inconclusive. Pertaining to verification, this means that if \(\phi\) encodes the violating traces of a program, \(\overline{\phi}\) represents fewer violating traces. The satisfiability of \(\phi\) entails the existence of \(\overline{\phi}\) a faulty trace in the original program while nothing

\(^2\)In this thesis, we use the term *abstraction* as a synonym for approximation (i.e. over- or underapproximation). In the literature, however, abstraction sometimes refers to overapproximations only.
can be concluded if it is unsatisfiable. Consequently, underapproximations can be used to prove an error, viz., the presence of a violating trace.

Approximation has been successfully applied to a large variety of decision problems (e.g. the satisfiability of bit-vector arithmetic [13, 12] and quantifier-free Presburger arithmetic [49]). To improve the performance of the decision procedures, the challenge is to find “good” approximations, that is, those which remove details in a formula that are likely to be irrelevant for the outcome of the decision procedure.

A popular technique for deriving approximations is based on *Craig interpolation* which, given a valid implication, is the problem of finding a particular formula called *interpolant* defined as follows:

**Definition 2.3.1** (Craig interpolant). *Given two formulae $A$ and $C$ such that $A \rightarrow C$ is valid, a formula $I$ is said to be an interpolant if:* 

1. $A \rightarrow I$ is valid
2. $I \rightarrow C$ is valid, and
3. $I$ only contains non-logical symbols occurring in both $A$ and $C$.

The existence of interpolants for any two first-order formulae $A$ and $C$ such that $A \rightarrow C$ is valid was first proved by William Craig in 1957 [26]. Interpolation can be used to derive image overapproximations for finite-state systems or path invariants in the case of infinite-state systems (cf. Section 2.3.2). To be applicable in practice, however, interpolation algorithms need to *efficiently* compute interpolants. Yet, no such procedure exists for Presburger arithmetic. As is common in formal verification, we also consider interpolation for unsatisfiable conjunctions $A \land B$, which corresponds to $C = \neg B$ in Definition 2.3.1.

As abstractions can lead to incorrect results, sound verification techniques must detect spuriousness and refine approximations to eventually yield a correct outcome. As a consequence, approximation-based algorithms all feature a refinement strategy for the case where an approximation is found to be too coarse. The next section presents well-known frameworks that employ approximations to increase the scalability of software verification.
2.3. Verification Techniques

2.3.2 Existing Model Checking Techniques

**Bounded Model Checking (BMC).** Bounded model checking is a variation of model checking which restricts the exploration of the state space to execution paths up to a certain length [4]. Thus, the technique either provides a guarantee that the first $k$ execution steps do not violate the property or a counterexample of length at most $k$.

Let $I$ be a formula representing the initial states of a program, $T$ a formula encoding the transition relation of the program and $P$ the property that is to be verified. A bounded model checker unwinds the transition relation and the property up to a given bound $k$ to obtain a formula $\phi_k$ defined as

$$
\phi_k := I(s_0) \land T(s_0, s_1) \land T(s_1, s_2) \cdots \land T(s_{k-1}, s_k) \land \left( \bigvee_{i=0}^{k} \neg P(s_i) \right)
$$

Figure 2.1: Overview of the bounded model checking procedure
where the subformula $T(s_{i-1}, s_i)$ evaluates to $True$ if and only if there is a transition from state $s_{i-1}$ to state $s_i$. Similarly, the formula $P(s_i)$ (resp. $I(s_i)$) evaluates to $True$ if and only if the state $s_i$ satisfies the property (resp. is an initial state). Consequently, the formula $\phi_k$ is satisfiable if and only if there is a trace of length $k$ that refutes the property.

Formula $\phi_k$ is then passed to a decision procedure. In the case of satisfiability, the decision procedure constructs a counterexample violating the property $P$. In the case of unsatisfiability, nothing can be concluded on the correctness of the program since faulty traces longer than $k$ may exist. In this case, the procedure starts again with the an increased bound $k'$. An overview of the general BMC procedure is depicted in Figure 2.1.

Clearly, if the program satisfies the property, the procedure does not terminate, since $k$ is increased infinitely often. Bounded model checking becomes complete, however, if $k$ exceeds the so called completeness threshold which is the minimum number of steps required for reaching all reachable states [50]. Computing the completeness threshold, however, is at least as hard as the general model checking problem [22].

Bounded model checking has been implemented in numerous tools for software verification. In [27], Currie et al. present the first implementation of a bounded model checker for software. As for programs written in C, the tool CBMC [23, 21] uses bit-flattening in order to reduce the satisfiability problem of $\phi_k$ to a satisfiability problem in propositional logic. Another variant of BMC was proposed in [1]. The implementation essentially expresses $\phi_k$ as a formula in integer linear arithmetic which is passed to an SMT-solver.

Counterexample-guided abstraction refinement (CEGAR). Counterexample-guided abstraction refinement is an iterative abstraction refinement technique [20]. The procedure generates an overapproximation of the system and refines the abstraction in the case where a spurious counterexample is found. To this end, the counterexample is simulated on the original system to detect if it indeed violates the property. If the trace is spurious, it is analyzed in order to extract information on how to refine the overapproximation. The procedure terminates if it can be ensured that the refinement process eventually leads to the original program.

Figure 2.2 depicts the CEGAR approach in more detail. The method relies on a
satisfiability checker capable of returning a counterexample (satisfying structure) in case satisfiability is detected. Assuming that $\phi$ is a formula encoding the program to be verified, the procedure takes $\phi$ as input and proceeds in four steps, namely abstraction, satisfiability checking, simulation and refinement.

1. **Abstraction.** A procedure $\mathcal{A}$ transforms the formula $\phi$ into an overapproximation $\overline{\phi}$ using an initially given set of transformations.

2. **Satisfiability checking.** The overapproximation $\overline{\phi}$ is verified for satisfiability using an appropriate decision procedure. If no (abstract) counterexample is returned, the verification process stops since the absence of errors for an overapproximation implies the correctness of the original program.

3. **Simulation.** The abstract counterexample is checked for feasibility in the original program. To this end, the program is symbolically simulated to determine whether the abstract counterexample also violates the property.
in the original program. If so, the verification stops and reports program incorrectness.

4. **Refinement.** The spurious (abstract) counterexample is analyzed to refine the overapproximation. This is done by identifying new predicates that prevent the abstract counterexample (and potentially others) from being discovered again, thus improving the accuracy of the overapproximation. The procedure is then called recursively on the refined abstraction.

**Alternating abstraction refinement.** The alternating approach by Bryant et al. [12] integrates both over- and underapproximations in an iterative refinement procedure. In [12] the technique was implemented for integer bit-vector arithmetic yielding a complete procedure for that theory. The procedure proceeds by generating either an over- or an underapproximation of a given formula. If the satisfiability check of the approximated formula is inconclusive, information obtained from the satisfiability checker is used to generate a refined approximation of the opposite type, and the procedure repeats. In this way, the approach alternates between over- and underapproximations to effectively prove or disprove correctness. We describe the procedure in more detail next.

Figure 2.3 presents an overview of the alternating abstraction approach. We assume that the formula \( \phi \) encodes a program in a given mathematical theory and that the satisfiability checker is capable of returning an unsatisfiability core in case it reports unsatisfiability. The procedure consists of the following four steps:

1. **Overapproximation.** Given an unsatisfiability core \( \text{core} \) (which is initially empty), the algorithm \( \mathcal{A} \) applies a predefined set of transformations on the original formula \( \phi \) to generate an overapproximation \( \overline{\phi} \). The role of the unsatisfiability core is to detect transformations leading to spurious results in subsequent iterations.

2. **Satisfiability checking.** The formula \( \overline{\phi} \) is checked for satisfiability. If the overapproximation \( \overline{\phi} \) is unsatisfiable, the procedure terminates and outputs program safety. Otherwise, a counterexample \( \alpha \) (satisfying structure) is extracted by the decision procedure.

3. **Underapproximation.** Given a satisfying structure \( \alpha \), procedure \( \mathcal{A} \) computes an underapproximation \( \phi \) using a predefined set of underapproximating transformations. The counterexample \( \alpha \) is used to find transformations
2.3. Verification Techniques

which lead to spurious results and thus, are not applied anymore.

4. Satisfiability checking. The underapproximation \( \phi \) is checked for satisfiability. In the case where satisfiability is returned, the procedure terminates and returns a safety violation. Otherwise an unsatisfiability core \( \text{core} \) is determined and the procedure proceeds incrementally.

Model checking by lazy abstraction with interpolants. Lazy interpolation-based model checking is a technique presented in [59] to verify infinite-state sequential programs. It proceeds by incrementally unwinding the control-flow graph (CFG) of a program to a tree. Figure 2.4 gives an overview of the verification procedure. Whenever a path from the program entry point to a program assertion (i.e. a safety property) is found, a verification condition

\[
\phi := T_1 \land T_2 \land \cdots \land T_n \land \neg P
\]
is generated, in which each $T_i$ corresponds to the $i$-th statement on an execution path of length $i$ and $P$ is the safety property to be verified. If $\phi$ is satisfiable, program incorrectness is reported. Otherwise, from a proof of unsatisfiability of the path $\phi$, it is then possible to derive a chain of interpolants:

**Definition 2.3.2 (Interpolant chain).** Given is a sequence of formulae $(F_1, \cdots, F_m)$ such that $\bigwedge_{i=1}^{m} F_i$ is unsatisfiable. A sequence of formulae $(I_0, \cdots, I_m)$ is said to be an interpolant chain iff

1. $I_0 = \text{true}$ and $I_m = \text{false}$,
2. $I_i \land F_{i+1} \rightarrow I_{i+1}$ is valid for all $i \in \{0, \ldots, m-1\}$,
3. $I_i$ only contains non-logical symbols occurring in both $T_i$ and $T_{i+1}$.

In other words, for the sequence $(T_1, \cdots, T_n, \neg P)$, each formula $I_i$ represents an intermediate path invariant. The interpolants $I_i$ are used to label the nodes of the program unwinding tree and are candidates for inductive path invariants. To check whether the interpolants actually are inductive, the notion of a covering...
relation is introduced, which is a binary relation between nodes of the unwinding tree. If all leaves of the unwinding tree are covered, the procedure halts reporting program safety. Otherwise, the unwinding tree is further unwound and the procedure is called recursively. As program safety is undecidable in general, the procedure might not terminate.
Craig interpolation has become a versatile tool in formal verification, used for instance, to generate intermediate assertions for safety analysis of programs. In this chapter, we consider Craig interpolation for full quantifier-free Presburger arithmetic (PA), for which currently no efficient interpolation procedures are known. Closing this gap, we introduce an interpolating sequent calculus for PA and prove it to be sound and complete. We have extended the PRINCESS theorem prover to generate interpolating proofs, and applied it to a large number of publicly available linear integer arithmetic benchmarks. The results indicate the robustness and efficiency of the proof-based interpolation procedure presented.

3.1 Motivation

In software verification, interpolation (cf. Definition 2.3.1) is applied to formulae encoding the transition relations of a model underlying a program (cf. Section 2.3.2). In order to support expressive programming languages, a great deal of effort has been invested in the design of algorithms that compute interpolants for formulae of various theories. As a result, efficient interpolation methods are available for propositional logic, linear arithmetic over the reals with uninter-
interpreted functions [58, 3, 76], data structures like arrays and sets [48], and other theories. As for integer arithmetic, a theory particularly relevant to software, interpolating solvers have so far been reported only for restricted fragments such as difference-bound logic, and logics with linear equalities and constant-divisibility predicates. For these theories, an interpolant can be derived in time polynomial in the size of the input formulae.

In this chapter, we push the boundaries of interpolation-based software model checking by presenting an interpolation method for full quantifier-free Presburger arithmetic (PA), i.e., linear arithmetic over the integers. This theory has been used to model the behavior of infinite-state programs and of hardware designs among other applications. Presburger arithmetic was shown to be decidable by quantifier elimination [65]. A brute-force interpolation method is to quantify out the variables not common to the input formulae, and then to eliminate those quantifiers. This approach suffers, however, from the triply-exponential complexity of the elimination procedure and tends to be ineffective in many practical cases.

A more promising approach (that has also been used in [58, 3, 54, 44] for example) is to extract interpolants directly from an unsatisfiability proof for $A \land B$. To this end, we first present a sound and complete proof system for PA based on a sequent calculus. We then augment the proof rules with labeled formulae and partial interpolants — proof annotations that, at the root of a closed proof, reduce to interpolants. In practice, the resulting interpolating proof system can be used to extend an existing unsatisfiability proof to one that interpolates. It can also serve as a replacement for the non-interpolating proof system, allowing the calculation of an interpolant on the fly. We prove our interpolating calculus to be sound and complete for PA. Our completeness result states that, for any valid implication, there exists a proof of its validity in our calculus, and the proof can be annotated with partial interpolants satisfying the proof rules.

In the case of PA, the primary difficulty when extracting interpolants from a proof is the treatment of mixed cuts: applications of a cut-rule (such as Gomory cuts [77] or the Omega rule (cf. Section 2.2.2 and [66])) to inequalities that have been derived as linear combinations of inequalities from both $A$ and $B$. Our work extends earlier interpolation procedures for linear arithmetic, in particular [54, 58], by defining an interpolating cut-rule called STRENGTHEN that can handle even mixed cuts. The rule subsumes a variety of cut-rules for integer linear programming, including Gomory cuts and the OMEGA-ELIM' rule (cf. Sec-
3.1. Motivation

A Motivating Example. Consider the following program with variables ranging over unbounded integers:

```plaintext
if (a == 2*x && a >= 0) {
    b = a / 2; c = 3*b + 1; assert (c > a); }
```

We would like to verify the assertion in the program. To this end, the program is translated into the PA formula below. Note that `b = a / 2` is converted into a
conjunction of two inequalities, and that the assertion is negated:

\[
\begin{align*}
    a - 2x &\equiv 0 \land -a \leq 0 \land 2b - a \leq 0 \land -2b + a - 1 \leq 0 \land \\
    c - 3b - 1 &\equiv 0 \land c - a \leq 0
\end{align*}
\] (3.1)

The unsatisfiability of (3.1) implies that no run of the program violates the assertion. Figure 3.1 shows a refutation of (3.1) in the Gentzen-style sequent calculus used in this chapter (the right-hand side \( \Delta \) happens to be empty in all sequents, which is not true in general). We add the prime symbol \( ' \) to the rule names to distinguish them from the interpolating rules introduced later. The proof starts with the conjunction (3.1) in the bottom sequent of the tree. Repeatedly applying the rule AND-LEFT\('' (denoted AND-LEFT\('')) splits the conjunction into a list of arithmetic literals. The equality \( a - 2x \equiv 0 \) is used to reduce the inequalities \(-a \leq 0, -2b + a - 1 \leq 0, \) and \( c - a \leq 0 \) by means of substitution (rule RED\(')). Similarly, \( c - 3b - 1 \equiv 0 \) is used to reduce \( c - 2x \leq 0 \). The inequalities \(-2x \leq 0\) and \(-2b + 2x - 1 \leq 0\) are simplified (rule SIMP\(')) by eliminating the coefficient \(2\); in the latter inequality, this requires rounding. Unsatisfiability of the remaining inequalities follows from two applications of the Fourier-Motzkin rule FM-ELIM\('), and the proof can be closed.

Interpolants for unsatisfiable formulae like (3.1) can reveal additional information about the program being investigated, for instance intermediate assertions. Suppose we want to compute an invariant for the program point immediately after \( b = a / 2 \). Let \( A \) denote the part of equation (3.1) encoding the program up to this point. Currently, the only known interpolation method for PA is to quantify out the local variables, i.e., variable \( x \) from \( A \):

\[
\exists x. \big( a - 2x \equiv 0 \land -a \leq 0 \land 2b - a \leq 0 \land -2b + a - 1 \leq 0 \big),
\]

which simplifies via quantifier elimination (QE) to \(-a \leq 0 \land 2b - a \equiv 0 \). Existentially quantifying out the local variables from \( A \) (or, universally, the local variables from the remaining part of (3.1)) always returns the strongest (respectively, weakest) interpolant for an unsatisfiable formula. These “extremal” interpolants may be very large, however. Suppose we modify the conditional in the program by adding further conjuncts that are unnecessary for the safety of the program:

\[
\text{if (a == 2*x && a >= 0 && a >= n*y - n/2 && a <= n*y) } \ldots
\] (3.2)
where \( n \in 2\mathbb{Z} \) is a parameter. The strongest (quantifier-free) interpolant, denoted \( I^n_s \), grows linearly in \( n \) and thus exponentially in the size of the program:

\[
I^n_s \equiv -a \leq 0 \land 2b - a \equiv 0 \land (n \mid a \land n \mid (a+1) \lor \cdots \lor n \mid (a + \frac{n}{2})).
\]

A weaker but much more succinct interpolant is the inequality \(-3b + a \leq 0\). We demonstrate in this chapter that proof-based interpolation provides a way of obtaining such succinct interpolants. Proofs can compactly encode the unsatisfiability of a formula and abstract away from irrelevant facts, enabling the extraction of succinct interpolants; this is of particular importance for program verification, where interpolants carrying unnecessary details can delay or prevent the discovery of inductive invariants (e.g., [59]). We therefore propose to lift proofs of unsatisfiability to interpolating proofs. This way, we avoid many disadvantages of QE-based interpolation, namely (i) its high complexity, (ii) its inflexibility in always returning a strongest or weakest interpolant, and (iii) the need to restart from scratch in order to consider a new partitioning of the unsatisfiable formula into \( A \) and \( B \) (in contrast, a proof-based method can extract many interpolants from a single proof).

### 3.2 An Interpolating Sequent Calculus for PA

Presburger arithmetic is the first order theory of linear integer arithmetic (see Section 2.1.2). For simplicity, we only allow 0 as the right-hand side of equalities and inequalities. Simultaneous substitution of a vector of terms \( \vec{t} = (t_1, \ldots, t_n) \) for variables \( \vec{x} = (x_1, \ldots, x_n) \) in a formula \( \phi \) is denoted by \( \left[ \vec{x}/\vec{t} \right] \phi \); we assume that variable capture is avoided by renaming bound variables as necessary. As short-hand notation, we sometimes also quantify over constants (as in \( \forall c. \phi \)) and assume that the constants are implicitly replaced by fresh variables. For reasons of presentation, we further assume that terms \( t \) are implicitly simplified to 0 or to the form \( \alpha_1 t_1 + \cdots + \alpha_n t_n \), in which \( 0 \notin \{\alpha_1, \ldots, \alpha_n\} \), and \( t_1, \ldots, t_n \) are pairwise distinct variables, constants, or 1.

**Interpolating sequents.** To extract interpolants from unsatisfiability proofs of \( A \land B \), formulae are labeled either with the letter \( L \) (“left”) to indicate that they are derived from \( A \), or with \( R \) (“right”) for formulae derived from \( B \). Formulae in
interpolating sequents are labeled either with the letter \( L \) to indicate that they are derived purely from \( A \), the letter \( R \) for formulae derived purely from \( B \), or with partial interpolants (PIs) that record the \( A \)-contribution to a formula obtained jointly from \( A \) and \( B \). Similarly as in [34], the labels \( L/R \) will be used to handle analytical rules that operate only on subformulae of the input formulae, while rewriting rules for arithmetic may mix parts of \( A \) and \( B \) and therefore require partial interpolants (as in [58]).

More formally, if \( \phi \) is a formula and \( t, t^A \) are terms, all without free variables, then \( \lfloor \phi \rfloor_L \) and \( \lfloor \phi \rfloor_R \) are \( L/R \)-labeled formulae and \( t = 0 \)[\( t^A = 0 \), \( t = 0 \)[\( t^A \neq 0 \)], and \( t \leq 0 \)[\( t^A \leq 0 \)] are formulae labeled with the partial interpolants \( t^A = 0 \), \( t^A \neq 0 \), and \( t^A \leq 0 \), respectively. We formally define interpolating sequents as follows

**Definition 3.2.1.** If \( \Gamma, \Delta \) are sets of labeled formulae and \( I \) is an unlabeled formula then \( \Gamma \vdash \Delta \upharpoonright I \) is an interpolating sequent if

(i) none of the formulae contains free variables,
(ii) \( \Gamma \) only contains formulae \( \lfloor \phi \rfloor_L, \lfloor \phi \rfloor_R, t = 0 \)[\( t^A = 0 \), \( t = 0 \)[\( t^A \neq 0 \)], and
(iii) \( \Delta \) only contains formulae \( \lfloor \phi \rfloor_L, \lfloor \phi \rfloor_R, t = 0 \)[\( t^A = 0 \), \( t = 0 \)[\( t^A \neq 0 \)].

Similarly, if \( \Gamma, \Delta \) are finite sets of unlabeled formulae without free variables, then \( \Gamma \vdash \Delta \) is an (ordinary) sequent. A sequent is valid if the formula \( \land \Gamma \rightarrow \lor \Delta \) is valid.

The semantics of interpolating sequents is defined with the help of projections \( \Gamma_L := \{ \phi \mid \lfloor \phi \rfloor_L \in \Gamma \} \) and \( \Gamma_R := \{ \phi \mid \lfloor \phi \rfloor_R \in \Gamma \} \) that extract the \( L/R \)-parts of a set \( \Gamma \) of labeled formulae.

**Definition 3.2.2.** An interpolating sequent \( \Gamma \vdash \Delta \upharpoonright I \) is valid if

(i) the sequent \( \Gamma_L \vdash I, \Delta_L \) is valid,
(ii) the sequent \( \Gamma_R, I \vdash \Delta_R \) is valid, and
(iii) the constants in \( I \) occur in both \( \Gamma_L \cup \Delta_L \) and \( \Gamma_R \cup \Delta_R \).

Note that formulae annotated with PIs are irrelevant for deciding whether an interpolating sequent is valid; this only depends on \( L/R \)-formulae. The semantics of PIs is made precise in Section 3.2.3; intuitively, a labeled formula \( \phi [\phi^A] \) in an interpolation problem \( A \land B \) expresses the implications \( A \rightarrow \phi^A \) and \( B \land \phi^A \rightarrow \phi \).

As special cases, \( [A]_L \vdash [C]_R \upharpoonright I \) reduces to \( I \) being an interpolant of the implication \( A \rightarrow C \), while \( [A]_L, [B]_R \vdash \upharpoonright I \) captures the concept of interpolants \( I \) for conjunctions \( A \land B \) common in formal verification.
Interpolating sequent calculi. An interpolating rule is a binary relation between a finite set of interpolating sequents called the premises, and a sequent called the conclusion.

\[ \Gamma_1 \vdash \Delta_1 \rightarrow I_1 \quad \cdots \quad \Gamma_n \vdash \Delta_n \rightarrow I_n \]

\[ \Gamma \vdash \Delta \rightarrow I \]

An interpolating rule is sound if, for all instances whose premises \( \Gamma_1 \vdash \Delta_1 \rightarrow I_1 \), \( \cdots \), \( \Gamma_n \vdash \Delta_n \rightarrow I_n \) are valid, the conclusion \( \Gamma \vdash \Delta \rightarrow I \) is valid, too. Interpolating proofs are trees growing upwards, in which each node is labeled with an interpolating sequent, and each non-leaf node is related to the node(s) directly above it through an instance of a calculus rule. A proof is closed if it is finite and all leaves are justified by an instance of a rule without premises.

To construct a proof for an interpolation problem, we build a proof tree starting from the root \( \Gamma \vdash \Delta \rightarrow I \) with unknown interpolant \( I \), i.e., \( I \) acts as a place holder. For example, to solve an interpolation problem \( A \land B \), we start with the sequent \( [A]_L, [B]_R \vdash \emptyset \rightarrow I \). Rules are then applied successively to decompose and simplify the sequent. Once all branches are closed, i.e. a proof is found, an interpolant can be extracted from the proof. Starting from the leaves, (intermediate) interpolants are computed and propagated back to the root leading to an interpolant \( I \). An example of an interpolating sequent proof is given next.

Example 3.2.3. We illustrate the concept of interpolating sequents with the proof in Figure 3.2, which is the interpolating version of the proof in Figure 3.1 and will serve as a running example in the whole section. For sake of brevity, we omit the subproofs \( \mathcal{A} \) and \( \mathcal{B} \). Due to the soundness of the applied calculus (stated in Section 3.2.3), the root sequent of the proof is valid, which implies that \( I_2 \equiv (-3b + a \leq 0) \) is an interpolant for the unsatisfiable conjunction (3.1). Note that \( I_2 \) is the inequality discussed in the motivating example of Section 3.1 as a succinct interpolant and intermediate program assertion.

3.2.1 Propositional, Initialization, and Closure Rules

To construct a proof for an interpolation problem \( A \land B \), we start with an interpolating sequent \( [A]_L, [B]_R \vdash \rightarrow I \) that contains only \( L/R \)-labeled formulae and apply propositional rules (middle and bottom part of Figure 3.3) to decompose \( A \) and \( B \) (e.g. the applications of rule AND-LEFT in Figure 3.2). When splitting over
Figure 3.2: The interpolating version of Figure 3.1. The initial interpolant generated by CLOSE-INEQ is \( I_1 = (-6b + 2a \leq 0) \equiv (-3b + a \leq 0) \), which is by STRENGTHEN combined with the interpolants \( false \) and \( \phi \) from the subproofs \( A \) and \( B \) to form the final interpolant \( I_2 = (I_1 \lor (false \land \phi)) \equiv I_1 \).

L-disjunctions in the antecedent (OR-LEFT-L), it is necessary to form the disjunction of the interpolants derived in the subproofs. Analogously, R-disjunctions yield conjunctive interpolants. All propositional rules propagate the \( L/R \)-label of formulae to their subformulae, unchanged. For brevity, we have omitted rules to move inequalities from the succedent to the antecedent.

Once the decomposition of formulae results in arithmetic literals, the initialization rules in the upper part of Figure 3.3 are used to turn \( L/R \)-formulae into formulae with PIs, to prepare them for later rewriting (the applications IPI in Figure 3.2). Generally, PIs for \( L \)-literals are chosen to be the literals themselves, while empty PIs are introduced for \( R \)-literals: the intuition is that \( L \)-formulae are fully contributed by \( A \), while \( R \)-formulae do not contain any \( A \)-contribution at all.

We observe that the IPI rules do not remove the \( L/R \)-formula to which they are applied (the formula occurs both in the conclusion and in the premise). The reason is that \( L/R \)-formulae in sequents track the vocabulary of symbols occurring in the input formulae \( A, B \); the vocabulary is used in condition (iii) of the definition of valid interpolating sequents, but also in the closure rules discussed
next. For completeness, it is never necessary to apply IPI rules twice on a proof branch to the same \(L/R\)-formula.

Finally, once rewriting (discussed in Section 3.2.2) has produced an unsatisfiable literal in an antecedent (or a valid literal in a succedent), a closure rule can be used to close the proof branch and to derive an interpolant from the PI of the unsatisfiable literal (the application \textsc{close-ineq} in Figure 3.2). Closure rules are given in the lower part of Figure 3.3. Because PIs can still contain local symbols that occur only in \(\Gamma_L \cup \Delta_L\) (and are not allowed in interpolants), it may be necessary to introduce existential quantifiers at this point. We note, however, that quantifiers in quantified literals can be eliminated in polynomial time; e.g., the quantified formula \(\exists c_1, \ldots, c_n. \alpha_1 c_1 + \cdots + \alpha_n c_n + t = 0\) is equivalent to the divisibility judgement \(\gcd(\alpha_1, \ldots, \alpha_n) \mid t\).

### 3.2.2 Rewriting Rules for Equality, Inequality and Divisibility

The arithmetic rewriting rules form a calculus to solve systems of equalities by means of Gaussian elimination and Euclid’s algorithm (the middle part of Figure 3.4), as well as a calculus for systems of inequalities based on Fourier-Motzkin elimination and cutting planes (the lower part of Figure 3.4). Decision procedures for PA in terms of the corresponding non-interpolating rules have been introduced in [71, 73] and directly carry over to the interpolating case. We therefore focus on the differences between the normal and the interpolating rules.

The rules \textsc{red-left/right} rewrite (in)equalities with equalities in the antecedent; in both cases, PIs are simply propagated along with the literals (\textsc{red-left} is applied repeatedly in Figure 3.2). The \textsc{red} rules alone do not form a complete calculus for integer equalities and have to be complemented with \textsc{col-red-L/R} to introduce fresh constants defined in terms of existing constants (the rules resemble column reductions when encoding systems of equalities as matrices). In combination, \textsc{red} and \textsc{col-red} are able to simulate the equality elimination procedure in [66], as well as standard procedures to transform sets of equalities (or matrices) to Hermite and Smith normalform [47, 44]. Because \textsc{col-red-L/R} only introduce local \(L/R\)-constants, it is guaranteed that the new constants do not occur in interpolants.

In contrast to [71, 73], we do not introduce a simplification rule \textsc{simp}' for literals, as full simplification is not always possible in the presence of PIs. For in-
Figure 3.3: The initialization, closure and propositional rules. The symbol $\exists_{LA}$ denotes existential quantification over constants that occur in $\Gamma_L, \Delta_L$ but not in $\Gamma_R, \Delta_R$, and $D \in \{L, R\}$. An equality is unsatisfiable iff it is of the form $\alpha_1 d_1 + \cdots + \alpha_n d_n + \alpha_0 \equiv 0$ and $\gcd(\alpha_1, \ldots, \alpha_n) \nmid \alpha_0$ (with $\gcd() := 0$).
3.2. An Interpolating Sequent Calculus for PA

\[
\begin{align*}
\Gamma, t \vdash 0 [t^A \vdash 0], s + \alpha \cdot t \vdash 0 [s^A + \alpha \cdot t^A \circ 0] & \vdash \Delta \triangleright I & \text{RED-LEFT} \\
\Gamma, t \vdash 0 [t^A \vdash 0], s \circ 0 [s^A \circ 0] & \vdash \Delta \triangleright I & \text{RED-RIGHT} \\
\Gamma, t \vdash 0 [t^A \vdash 0] & \vdash s + \alpha \cdot t \vdash 0 [s^A + \alpha \cdot t^A \circ 0], \Delta \triangleright I & \text{MUL-LEFT} \\
\Gamma, t \circ 0 [t^A \circ 0] & \vdash \Delta \triangleright I & \text{MUL-RIGHT} \\
\Gamma, u - c \vdash 0 [u^A - d \vdash 0] & \vdash \Delta \triangleright I & \text{COL-RED} \\
\Gamma, [\exists x. \alpha x + t \vdash 0]_D & \vdash \Delta \triangleright I & \text{DIV-LEFT} \\
\Gamma, [\alpha | t]_D & \vdash \Delta \triangleright I & \text{DIV-RIGHT} \\
\Gamma, s \leq 0 [s^A \leq 0], t \leq 0 [t^A \leq 0], \alpha s + \beta t \leq 0 [\alpha s^A + \beta t^A \leq 0] & \vdash \Delta \triangleright I & \text{FM-ELIM} \\
\Gamma, t \vdash 0 [t^A \vdash 0] & \vdash \Delta \triangleright E & \text{STRENGTHEN} \\
\Gamma, t + 1 \leq 0 [t^A \leq 0] & \vdash \Delta \triangleright I^0 \\
\Gamma, t + 1 \leq 0 [t^A + 1 \leq 0] & \vdash \Delta \triangleright I^1 \\
\Gamma, t \leq 0 [t^A \leq 0] & \vdash \Delta \triangleright I^1 \lor (E \land I^0) & \text{SPLIT-EQ} \\
\Gamma, t + 1 \leq 0 [t^A + 1 \leq 0] & \vdash \Delta \triangleright I \\
\Gamma, -t + 1 \leq 0 [-t^A + 1 \leq 0] & \vdash \Delta \triangleright J \\
\Gamma, t + 1 \leq 0 [t^A \leq 0] & \vdash \Delta \triangleright J \\
\Gamma, t + 1 \leq 0 [t^A \leq 0] & \vdash \Delta \triangleright J \\
\Gamma, t \vdash 0 [t^A \neq 0], \Delta \triangleright I \land J & \text{SPLIT-NEQ}
\end{align*}
\]

Figure 3.4: Rules for equality/divisibility and inequality constraints. In COL-RED, \(c, d\) are constants occurring in neither the conclusion, \(u\) nor \(u^A\). The term \(u^A\) is such that \(u^A\) (resp. \(u - u^A\)) only contains constants from \(\Gamma_L \cup \Delta_L\) (resp. \(\Gamma_R \cup \Delta_R\)). In all rules \(\alpha, \beta \in \mathbb{Z}_{>0}, \circ \in \{\div, \leq\}, \odot \in \{\div, \neq\}\) and \(D \in \{L, R\}\).
### Partial interpolant annotation

<table>
<thead>
<tr>
<th>Γ, t = 0 [r^A = 0] ⊢ Δ</th>
<th>Γ_L ⊢ r^A = 0, Δ_L</th>
<th>Γ_R ⊢ t - r^A = 0, Δ_R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Γ, t ≤ 0 [r^A ≤ 0] ⊢ Δ</td>
<td>Γ_L ⊢ r^A ≤ 0, Δ_L</td>
<td>Γ_R ⊢ t - r^A ≤ 0, Δ_R</td>
</tr>
<tr>
<td>Γ ⊢ t = 0 [r^A = 0], Δ</td>
<td>Γ_L ⊢ r^A = 0, Δ_L</td>
<td>Γ_R ⊢ t - r^A = 0, Δ_R</td>
</tr>
<tr>
<td>Γ ⊢ t = 0 [r^A ≠ 0], Δ</td>
<td>Γ_L ⊢ r^A = 0, Δ_L</td>
<td>Γ_R, t - r^A = 0 ⊢ Δ_R</td>
</tr>
</tbody>
</table>

Table 3.1: Correctness conditions (1) and (2) for partial interpolants

In practice, the equality 2x = 0 [a = 0] cannot be simplified to the form x = 0 [r^A = 0] (as it would happen in [71, 73]) because the factor 2 does not occur in the PI. This raises a potential problem, as terms αx cannot be rewritten to 0 with the help of 2x = 0 if α is odd. As a solution, we introduce the rules MUL-LEFT/RIGHT to multiply terms with positive integers prior to rewriting.

Similar to rewriting with equalities, inequalities can be added to each other with the help of the rule FM-ELIM realizing Fourier-Motzkin variable elimination. The STRENGTHEN rule is introduced to achieve completeness over the integers (Figure 3.2 shows applications of FM-ELIM and STRENGTHEN). Compared to the calculi in [71, 73], the use of STRENGTHEN in our interpolating calculus is threefold: (i) STRENGTHEN can simulate the OMEGA-ELIM’ rule the Omega test (cf. Section 2.2.2), (ii) as shown in Figure 3.2, repeated application of STRENGTHEN can be used to round inequalities αt + β ≤ 0 to αt + αfloor(β/α) ≤ 0 (which is done by SIMP’ in [71, 73]), and (iii) STRENGTHEN can simulate the law of anti-symmetry that is implemented by the rule ANTI-SYM’ in [71, 73]. As STRENGTHEN is the most central rule in our calculus, we provide a detailed discussion in Section 3.3.

### 3.2.3 Soundness and Completeness

**Soundness.** Our interpolating calculus generates correct interpolants: whenever a sequent [A]_L ⊢ [C]_R ⇒ I is derived, the implications A → I → C are valid, and all constants in I occur in both A and C. More generally:

**Theorem 3.2.4 (Soundness).** If an interpolating sequent Γ ⊢ ∆ ⇒ I without any
3.2. An Interpolating Sequent Calculus for PA

PIs is provable in the calculus, then it is valid. This implies, in particular, that the sequent \( \Gamma_L, \Gamma_R \vdash \Delta_L, \Delta_R \) is valid. (Proof pg. 107)

To prove this theorem, we first need to define the semantics of PIs (although the sequent \( \Gamma \vdash \Delta \rightarrow I \) in the theorem does not contain any PIs, they are likely to be introduced in the course of a proof). We say that a PI is correct if the sequents (1) and (2) given in Table 3.1 are valid, \( t^A \) only contains constants that occur in \( \Gamma_L \cup \Delta_L \), and \( t - t^A \) only contains constants that occur in \( \Gamma_R \cup \Delta_R \). Soundness is then proven in two steps: (i) We show that all PIs in a closed proof are correct by induction on the distance of a sequent from the root of the proof: assuming that all PIs in the conclusion of a rule application are correct, we prove that the PIs in the rule premises are correct. (ii) We show the validity of all sequents in a closed proof by induction on the size of sub-proofs: assuming that all premises of a rule are valid, we prove that the conclusion is valid, too.

As a technical difficulty, we need to annotate some rules by introducing further auxiliary formulae in the premises to ensure (i) holds. These annotations are only required for the soundness proof; soundness of the rules with auxiliary formulae directly implies soundness of the original rules. More details are given in Appendix B.1.1 on page 107.

Completeness. Vice versa, whenever an implication \( A \rightarrow C \) holds, our calculus is able to derive an interpolant.

Theorem 3.2.5 (Completeness). Suppose \( \Gamma, \Delta \) are sets of labeled formulae \( [\phi]_L \) and \( [\phi]_R \). If \( \Gamma_L, \Gamma_R \vdash \Delta_L, \Delta_R \) is valid, then there exists a formula \( I \) such that the interpolating sequent \( \Gamma \vdash \Delta \rightarrow I \) is provable. (Proof pg. 111)

The lemma follows from the completeness of the calculi in [71, 73] by means of proof lifting: given that \( \Gamma_L, \Gamma_R \vdash \Delta_L, \Delta_R \) is valid, there is a proof of this fact in the non-interpolating calculus. This proof can be lifted by replacing each rule application with an application of the corresponding interpolating rule.
3.3 Mixed Cuts and Strengthening

3.3.1 Mixed Cuts

Reasoning in linear integer arithmetic generally requires some kind of cut-rule to deal with the phenomenon of formulae that are satisfiable over the rationals, but unsatisfiable over integers. The non-interpolating calculus in [71] provides two rules for this: the \textsc{Simp'} rule to round inequalities \( \alpha t + \beta \leq 0 \) to \( \alpha t + \lceil \frac{\beta}{\alpha} \rceil \leq 0 \) (which resembles Gomory cuts [77]), and the general \textsc{Strengthen'} rule:

\[
\frac{\Gamma, t = 0 \vdash \Delta}{\Gamma, t + 1 \leq 0 \vdash \Delta} \quad \text{STRENGTHEN'}
\]

Because \textsc{Strengthen'} subsumes rounding via the rule \textsc{Simp'}, we can ignore the latter rule for the time being and concentrate on \textsc{Strengthen'}.

In order to lift \textsc{Strengthen'} to the interpolating calculus, we can first observe that two special cases are easy to handle:

\[
\frac{\Gamma, t = 0 \vdash \Delta \quad \Gamma, t + 1 \leq 0 \vdash \Delta}{\Gamma, t \leq 0 \vdash \Delta} \quad \text{STRENGTHEN-L}
\]

\[
\frac{\Gamma, t = 0 \vdash \Delta \quad \Gamma, t + 1 \leq 0 \vdash \Delta}{\Gamma, t \leq 0 \vdash \Delta} \quad \text{STRENGTHEN-R}
\]

These cases are called pure cuts in [54], because the PIs indicate that the inequality \( t \leq 0 \) has been derived only from \( L \)- or only from \( R \)-formulae, respectively. Strengthening inequalities of this kind corresponds to splitting a disjunction labeled with \( L \) or \( R \).

The general case is known as mixed cut [54] and encompasses an application of \textsc{Strengthen} to a formula \( t \leq 0 [t^A \leq 0] \) with \( t^A \notin \{0, t\} \); the rule for this general case is given in Figure 3.4 and features three premises, one more than the non-interpolating rule \textsc{Strengthen'}. To understand the shape of \textsc{Strengthen}, note that we can represent \( t \leq 0 \) as the sum of the inequalities \( t^A \leq 0 \) and \( t - t^A \leq 0 \), the first of which is derived from \( L \)-formulae, and the second from \( R \)-formulae. The effect of \textsc{Strengthen} can then be simulated by applying \textsc{Strengthen-L} to \( t^A \leq 0 [t^A \leq 0] \) and then \textsc{Strengthen-R} to \( t - t^A \leq 0 [0 \leq 0] \).

The combined application of the two rules explains the interpolant \( I^1 \lor (E \land I^0) \) resulting from \textsc{Strengthen}.
3.3. **Mixed Cuts and Strengthening**

**Complexity.** Non-interpolating refutations of unsatisfiable conjunctions of literals have exponential size in the worst case [77]. Similarly, it can be shown that any valid sequent (without quantifiers or propositional connectives) has interpolants of worst-case exponential size that can be derived using a proof of worst-case exponential size (using the rules **STRENGTHEN-L/R** from above).

In general, however, lifting a non-interpolating to an interpolating proof can increase the size of the proof exponentially, due to two reasons: (i) the fact that **STRENGTHEN** in Figure 3.4 has three premises, while the non-interpolating rule **STRENGTHEN'** has only two, which can make it necessary to repeatedly duplicate subproofs during lifting (this is partly addressed in Section 3.3.2), and (ii) because the rule **SIMP'** (which has to be simulated by **STRENGTHEN** in the interpolating calculus) often allows very succinct proofs. As a result, there are unsatisfiable conjunctions $A \land B$ with non-interpolating proofs of linear size, although all interpolants have exponential size.

### 3.3.2 Successive Strengthening

It is quite common that **STRENGTHEN** is applied repeatedly to a sequence $t \leq 0$, $t+1 \leq 0$, $t+2 \leq 0$, ... of inequalities, for instance to simulate rounding of an inequality or the **OMEGA-ELIM'** rule (cf. Section 2.2.2). Because each application of **STRENGTHEN** generates two new inequalities, $2^k - 1$ applications are necessary in order to strengthen an inequality $t \leq 0$ to $t + k \leq 0$, and the resulting interpolant will also be of exponential size. To tackle this growth, we present an optimized rule that captures $k$-fold strengthening and requires only a quadratic number of premises. The optimized rule **$k$-STRENGTHEN** exploits the fact that many of the goals created by repeated application of **STRENGTHEN** are redundant:

\[
\begin{align*}
\{ \Gamma, t + i \vdash 0 [t_A + j \vdash 0] \vdash \Delta \upharpoonright E_i^j \} & \quad 0 \leq j \leq i < k \\
\{ \Gamma, t + k \vdash 0 [t_A + j \vdash 0] \vdash \Delta \upharpoonright I_i^j \} & \quad 0 \leq j \leq k \\
\Gamma, t \vdash 0 [t_A \vdash 0] \vdash \Delta \upharpoonright K & \quad \text{**$k$-STRENGTHEN**}
\end{align*}
\]

where the resulting interpolant $K$ is defined by:

\[
K = \bigvee_{0 \leq j \leq k} \left( I_i^j \land \bigwedge_{j \leq i < k} E_i^j \right)
\]  
(3.3)
The size of $K$ grows quadratically, rather than exponentially, in $k$. Thus, whenever the STRENGTHEN rule is to be applied $k$ times in succession, it is possible and more efficient to use the $k$-STRENGTHEN rule instead.

The number of premises of $k$-STRENGTHEN (but not the size of the resulting interpolant) can be reduced further to a linear number: any two premises generating $E_i^j$ and $E_i^l$ differ only in the partial interpolant of $t + i \leq 0$, not in any other formula. We can exploit this by treating the family $(E_i^j)_{0 \leq j \leq i}$ as a single premise that is parameterized in the free variable $j$. This way, a single subproof can generate a parameterized interpolant $E_i^j(j)$. The parameter $j$ can be instantiated to the values $0 \leq j \leq i$ when constructing $K$. Parametrized interpolants $I(j)$ can be derived in a similar way.

**Lemma 3.3.1.** If the premises of the $k$-STRENGTHEN rule are valid interpolating sequents, then the interpolating sequent in the conclusion is valid. In particular this means, that $K$ is an interpolant for the sequent $\Gamma, t \leq 0 \vdash \Delta$. (Proof pg. 113).

### 3.3.3 Interpolation of Rounding Operations

An additional optimization is possible when the rule $k$-STRENGTHEN is used to round an inequality $\alpha t + \beta \leq 0$ to $\alpha t + \alpha \lceil \frac{\beta}{\alpha} \rceil \leq 0$. Rounding corresponds to $k$-STRENGTHEN with $k = \alpha \lceil \frac{\beta}{\alpha} \rceil - \beta$:

$$\begin{align*}
\{ \Gamma, \alpha t + \beta + i = 0 [t_A + j = 0] \vdash \Delta \rightarrow E_i^j \}_{0 \leq j \leq i < k} \\
\{ \Gamma, \alpha t + \alpha \lceil \frac{\beta}{\alpha} \rceil \leq 0 [t_A + j \leq 0] \vdash \Delta \rightarrow I^j \}_{0 \leq j \leq k}
\end{align*}$$

$$\Gamma, \alpha t + \beta \leq 0 [t_A \leq 0] \vdash \Delta \rightarrow K$$

$k$-STRENGTHEN

We can observe that $\alpha t + \beta + i = 0$ is unsatisfiable for $0 \leq i < \alpha \lceil \frac{\beta}{\alpha} \rceil - \beta$, so that the equality-premises can be closed immediately via CLOSE-EQ-LEFT. Consequently, the interpolants $E_i^j = E^j = (\exists_{t_A} t_A^k + j = 0)$ do not depend on $i$, and the overall interpolant can be simplified to $K = I^k \lor \bigvee_{0 \leq j < k} (I^j \land E^j)$.

**Example 3.3.2.** We use $k$-STRENGTHEN to compute an interpolant for the conjunction $A \land B$ with $A = -y + 5x - 1 \leq 0 \land y - 5x \leq 0$ and $B = 5z - y + 1 \leq 0 \land -5z + y - 2 \leq 0$. Note that $A \land B$ is satisfiable over rationals, but unsatisfiable over the integers. An interpolating proof of unsatisfiability is as follows:
3.4. Experimental Results

We implemented the proposed interpolating calculus on top of the PRINCESS theorem prover [73], including all optimizations described in Section 3.3. To this end, we extended PRINCESS to generate proofs. The interpolation procedure then processes the proof and generates an interpolant using the rules presented in this chapter. The benchmarks for our experiments are derived from the SMT-LIB category QF-LIA. We evaluate them on a 3 GHz Intel Pentium Xeon processor with 4 MB cache and running Linux. Because SMT-LIB benchmarks are usually conjunctions at the outermost level, we partitioned them into \( A \land B \) by choosing the first \( \frac{k}{10} \cdot n \) of the benchmark conjuncts as \( A \), the rest as \( B \) (where \( n \) is the total number of conjuncts, and \( k \in \{1, \ldots, 9\} \)). Partitionings where \( A \) did not contain any local symbols (constants or propositional variables) were ignored.

\[ K = \underbrace{3 - 1 \leq 0}_{I^k} \lor \bigvee_{0 \leq j < 3} \underbrace{(j - 1 \leq 0 \land 5 \mid (y + 1 - j))}_{E^j} \equiv 5 \mid (y + 1) \lor 5 \mid y \]

Most importantly, the rule 3-STRENGTHEN is used to round \( -5z + 5x - 3 \leq 0 \) to \( -5z + 5x \leq 0 \), from which a contradiction can be derived via FM-ELIM. In the premises of 3-STRENGTHEN, the inequality interpolants \( I^j = (j - 1 \leq 0) \) and the equality interpolants \( E^j = (\exists x. -y + 5x - 1 + j = 0) \equiv (5 \mid (y + 1 - j)) \) are derived as discussed above. The overall interpolant is:

\[ K \]

\[ K = \underbrace{3 - 1 \leq 0}_{I^k} \lor \bigvee_{0 \leq j < 3} \underbrace{(j - 1 \leq 0 \land 5 \mid (y + 1 - j))}_{E^j} \equiv 5 \mid (y + 1) \lor 5 \mid y \]

1Implementation and benchmarks: www.philipp.ruemmer.org/iprincess.shtml
Chapter 3. Interpolation for Presburger Arithmetic

Figure 3.5: Benchmarks comparing proof-based interpolation (PBI) with quantifier elimination (QE)

- PBI (runtime in seconds)
- QE (runtime in seconds)
- PBI (interpolant size)
- QE (interpolant size)
Since, to the best of our knowledge, no other interpolation procedure for PA was available, we compared the performance of the PRINCESS interpolation procedure with interpolation by quantifier elimination (QE), eliminating all local symbols in $A$. For the latter, we use the implementation of the Omega test [66] available in PRINCESS. The results are shown in Figure 3.5.

The upper left diagram compares runtimes of proof-based interpolation (PBI) with QE, with a timeout of 120s. We have not included the time to generate proofs, because in typical applications (like software model checking) many interpolants will be generated from each proof, and because QE does not decide the input formula. Considering only the cases without timeout, proving took on average about 4 times as long as the extraction of all interpolants from one proof. The diagram shows that PBI outperforms QE in 147 out of 205 cases, while QE is faster in 58 cases. QE times out for 103 of the benchmarks, PBI for 29. When analyzing the cases where QE is faster than PBI, we observed that QE typically performs well when $A$ only contains few local symbols, i.e., when few quantifiers need to be eliminated. We highlight cases where the number of local symbols is less than 15 by gray points in the diagrams; with an increasing number of local symbols, the performance of QE quickly degrades. To quantify this phenomenon, we measured interpolation runtimes classified by the number of local symbols in $A$: the two lower diagrams in Figure 3.5 show that PBI is a lot less dependent on the number of such symbols than QE.

The upper right diagram compares the sizes of the interpolants (the number of operators) generated by the two techniques. In 149 cases, the interpolants obtained using PBI are smaller than those derived by QE, in 122 cases they are at least one order of magnitude smaller.

## 3.5 Related Work

Interpolation for propositional logic, linear rational arithmetic, and uninterpreted functions is a thoroughly explored field. In particular, McMillan presents an interpolating theorem prover for rational arithmetic and uninterpreted functions [58]; an interpolating SMT solver for the same logic has been developed by Beyer et al. [3]. Rybalchenko et al. [76] introduce an interpolation procedure for this logic that works without constructing proofs.
Interpolation has also been investigated in several fragments of integer arithmetic. McMillan considers the logic of difference-bound constraints [59], which is decidable by reduction to rational arithmetic. As an extension, Cimatti et al. [17] present an interpolation procedure for the UTVPI fragment of linear integer arithmetic. Both fragments allow efficient reasoning and interpolation, but are not sufficient to express many typical program constructs, such as integer division. In [44], separate interpolation procedures for two theories are presented, namely (i) PA restricted to conjunctions of integer linear (dis)equalities and (ii) PA restricted to conjunctions of stride constraints. The combination of both fragments with integer linear inequalities is not supported, however.

Kapur et al. [48] prove that full PA is closed under interpolation (as an instance of a more general result about recursively enumerable theories), but their proof does not directly give rise to an efficient interpolation procedure. Lynch et al. [54] define an interpolation procedure for linear rational arithmetic, and extend it to integer arithmetic by means of Gomory cuts. No interpolating rule is provided for mixed cuts, however, which means that sometimes formulae are generated that are not true interpolants because they violate the vocabulary condition (i.e., contain symbols that are not common to $A$ and $B$).

### 3.6 Summary

We have presented the first interpolating sequent calculus for quantifier-free Presburger arithmetic, permitting arbitrary combinations of linear integer equalities, inequalities, and stride predicates. Our calculus is intended to be used with a reasoning engine for sequent calculi, resulting in an interpolating decision procedure for Presburger arithmetic. We have implemented our calculus rules in PRINCESS and demonstrated experimentally that our method is able to generate much more succinct interpolants than quantifier elimination, which is the only other method for Presburger interpolation we are aware of.

To integrate the presented interpolation procedure into a software model checker based on lazy abstraction [59] however, Presburger arithmetic is not sufficient. Arrays, integer machine division and multiplication, for example, are not expressible with pure quantifier-free theory of Presburger arithmetic (PA). The next chapter presents several theory extensions that are required to express such operations and data structures.
Interpolation for Extensions of Presburger Arithmetic

Extensions of quantifier-free Presburger arithmetic (PA) are necessary to encode program operations such as integer division and array manipulations. Therefore, to apply interpolation-based software verification techniques, the interpolation method described in Chapter 3 needs be enhanced to support extensions of PA. This chapter presents interpolation procedures for the extended theories of PA combined with (i) uninterpreted predicates (QPA+UP), (ii) uninterpreted functions (QPA+UF) and (iii) extensional arrays (QPA+AR). We demonstrate that none of these combinations can be effectively interpolated without the use of quantifiers, even if the input formulae are quantifier-free. For QPA+UP and QPA+UF we identify fragments with restricted forms of guarded quantification that are closed under interpolation, and that can easily be mapped to quantifier-free expressions in common programming languages. For QPA+AR, we formulate a sound interpolation procedure that potentially produces interpolants with unrestricted quantifiers.

4.1 Motivation

The goal of this chapter is an interpolation procedure that is instrumental in analysing programs manipulating integer variables. We therefore consider the first-
order theory of quantified Presburger arithmetic (linear integer arithmetic, QPA) combined with the theories of uninterpreted predicates (UP) and uninterpreted functions (UF). These combinations in turn allow us to encode the theory of extensional arrays (AR), using uninterpreted function symbols for read and write operations. Our interpolation procedure extracts an interpolant directly from a proof of $A \Rightarrow C$. Starting from a sound and complete proof system based on a sequent calculus, the proof rules are extended by labeled formulae and annotations that reduce, at the root of a closed proof, to interpolants. In earlier work, we presented a similar procedure for quantifier-free Presburger arithmetic [9].

In program verification, interpolating theorem provers often interact closelyly with model checkers [46, 60]. It is therefore essential that interpolants computed by the prover be expressible in the logic of the programming language. Unfortunately, interpolation procedures for expressive first-order fragments such as arithmetic with uninterpreted predicates, as we consider them here, often generate interpolants with quantifiers that cannot be mapped to expressions in the programming language. This is not by accident. In fact, in this chapter we first show that interpolation of QPA+UP in general requires the use of quantifiers, even if the input formulae are themselves free of quantifiers.

In order to solve this problem, we study fragments of QPA+UP that are closed under interpolation: fragments such that interpolants for input formulae can again be expressed in the theory. According to the result above, such fragments must allow at least a limited form of quantification. Our second contribution is to show that the theory PAID+UP of Presburger arithmetic with uninterpreted predicates and a restricted form of guarded quantifiers is indeed closed under interpolation. A similar fragment, PAID+UF, can be identified for the combination of Presburger arithmetic with uninterpreted functions. Moreover, the guarded quantifiers can be rewritten into quantifier-free integer divisibility predicates, making them readily expressible in commonly used programming languages such as C or Java.

In summary, this chapter presents an interpolating calculus for the first-order theory of Presburger arithmetic and uninterpreted predicates, QPA+UP. We show that, for some input formulae, quantifiers cannot be avoided in interpolants, and suggest a restriction of QPA+UP that is closed under interpolation, yet permits interpolants conveniently expressible in typical program logics. We extend these results to Presburger theories with uninterpreted functions and, specifically, to
4.2 Interpolation for Uninterpreted Predicates

4.2.1 Presburger Arithmetic and Uninterpreted Predicates

We begin by studying the interpolation problem for Presburger arithmetic extended with uninterpreted predicates (QPA+UP), which forms a simple yet expressive base logic in which functions and arrays can be elegantly encoded. The case of predicates is instructive, since essentially the same phenomena occur under interpolation as with uninterpreted functions.

**Example 4.2.1.** We illustrate the construction of an interpolating proof by deriving an interpolant for $A \Rightarrow C$, with $A = \neg p(c) \lor p(d) \land p(c)$ and $C = p(d)$. A complete interpolating proof of this implication reads as follows:

\[
\begin{array}{c}
\vdash [p(c)]_L \vdash [p(d)]_R, [p(c)]_L \nrightarrow false \\
\vdash [\neg p(c)]_L, [p(c)]_L \vdash [p(d)]_R \nrightarrow false \\
\vdash [\neg p(c) \lor p(d)]_L, [p(c)]_L \vdash [p(d)]_R \nrightarrow false \lor p(d) \\
\vdash [(\neg p(c) \lor p(d)) \land p(c)]_L \vdash [p(d)]_R \nrightarrow false \lor p(d) \\
\end{array}
\]

The shaded regions indicate the parts of the formula being matched against the rules in Figures 3.3 and 3.4. The sequent $[(p(c) \lor p(d)) \land p(c)]_L \vdash [p(d)]_R \nrightarrow I$ is the root of the proof; $I$ is to be filled in once the proof is closed. The AND-LEFT rule propagates the L-label to the sub-formulae of the antecedent of the first sequent. By applying OR-LEFT-L to the disjunction $p(c) \lor p(d)$, the proof splits into two branches. The right branch can immediately be closed using CLOSE-LR. The left branch requires an application of NOT-LEFT before it can be closed with CLOSE-LL. We compute an interpolant by propagating (intermediate) interpolants from the leaves back to root of the proof. As specified by CLOSE-LR, the interpolant of the right branch is $p(d)$. On the left branch, the CLOSE-LL rule yields the interpolant false, which is not modified by NOT-LEFT. The rule OR-LEFT-L takes the interpolants of its two subproofs and generates $false \lor p(d)$.
This is the final interpolant, since the last rule \textsc{and-left} does not modify interpolants.

In this example, the arguments of occurrences of uninterpreted predicates literally match up, which is not always the case. The rules presented so far are insufficient to prove more complex theorems, such as \( p(c) \land c \div d \rightarrow p(d) \), in which arithmetic and predicate calculus interact. To fully integrate uninterpreted predicates, we use an explicit \textit{predicate consistency} axiom

\[
PC_p = \forall \bar{x}, \bar{y}. ((p(\bar{x}) \land \bar{x} - \bar{y} \div 0) \rightarrow p(\bar{y}))
\]  

(4.1)

which can be viewed as an \textsc{L}- or \textsc{R}-labeled formula that is implicitly present in every sequent (depending on whether \( p \) occurs in \( \Gamma_L \cup \Delta_L \), \( \Gamma_R \cup \Delta_R \), or both).

To make use of (4.1) in a proof, we need additional proof rules to instantiate quantifiers, which are given in the Figure 4.1. Formula (4.1) can be instantiated with techniques similar to the \textsc{e-matching} in SMT solvers [29]: it suffices to generate a ground instance of (4.1) by applying \textsc{all-left-\textsc{L}/\textsc{R}} whenever literals \( p(\bar{s}) \) and \( p(\bar{t}) \) occur in the antecedent and succedent [73]:

\[
\begin{array}{c}
\frac{\Gamma, [p(\bar{s})]_D, [(p(\bar{s}) \land \bar{s} - \bar{t} \div 0) \rightarrow p(\bar{t})]_L \vdash [p(\bar{t})]_E, \Delta \triangleright I}{\Gamma, [p(\bar{s})]_D \vdash [p(\bar{t})]_E, \Delta \triangleright \forall_{R\bar{s}} I} \quad \text{\textsc{all-left-\textsc{L}/\textsc{R}}}^+ \\
\end{array}
\]

where \( D, E \in \{ L, R \} \) are arbitrary labels, and \( \forall_{R\bar{s}} \) denotes universal quantification over all constants occurring in the terms \( \bar{s}, \bar{t} \) but not in the set of left formulae \( (\Gamma, [p(\bar{s})]_D)_L \cup (\Delta, [p(\bar{t})]_E)_L \) (like in Figure 4.1). Similarly, instances of (4.1) labeled with \( R \) can be generated using \textsc{all-left-\textsc{R}}. To improve efficiency, refinements can be formulated that drastically reduce the number of generated instances [72].

\textbf{Correctness.} The calculus consisting of the rules in Figures 3.3, 3.4 and 4.1 as well as axiom (4.1) generates correct interpolants. That is, whenever a sequent \( [A]_L \vdash [C]_R \triangleright I \) is derived, the implications \( A \Rightarrow I \) and \( I \Rightarrow C \) are valid, and the constants and predicates in \( I \) occur in both \( A \) and \( C \). More precisely:

\textbf{Lemma 4.2.2 (Soundness).} If an interpolating QPA+UP sequent \( \Gamma \vdash \Delta \triangleright I \) is provable in the calculus, then it is valid.
4.2. INTERPOLATION FOR UNINTERPRETED PREDICATES

![Interpolating rules to handle quantifiers. Parameter D stands for either L or R. The quantifier ∀Rt denotes universal quantification over all constants occurring in t but not in ΓL ∪ ΔL; likewise, ∃Lt denotes existential quantification over all constants occurring in t but not in ΓR ∪ ΔR.]

In particular, the sequent ΓL, ΓR ⊢ ΔL, ΔR is valid in this case. As shown in [9], this lemma holds for the calculus consisting solely of the arithmetic and propositional rules. It is easy to see that the additional rules presented in this chapter are sound, too.

Concerning completeness, we observe that the logic of quantified Presburger arithmetic with predicates is $\Pi^1_1$-complete, which means that no complete calculi exist [40]. On the next pages, we therefore discuss how to restrict the quantification allowed in formulae to achieve completeness, while retaining the ability to extract interpolants from proofs.

4.2.2 Quantifiers in QPA+UP Interpolants

We first consider the quantifier-free fragment PA+UP. With the help of results in [73, 9], it is easy to see that our calculus is sound and complete for PA+UP, and can in fact be turned into a decision procedure. There is a caveat, however: although formulae in PA+UP are quantifier-free, generated interpolants may still contain quantifiers and thus lie outside of PA+UP. The source of quantifiers is the rules ALL-LEFT-L/R in Figure 4.1, which can be used to instantiate L/R-labeled quantified formulae with terms containing alien symbols. Such symbols have to be eliminated from resulting interpolants through quantifiers.

**Theorem 4.2.3.** PA+UP is not closed under interpolation. (Proof pg. 127)

In other words, there are unsatisfiable conjunctions $A \land B$ in PA+UP for which no interpolants expressible in PA+UP exist. The following example illustrates this
Example 4.2.4. Figure 4.2 shows the derivation of an interpolant for the unsatisfiable conjunction \((2c - y = 0 \land p(c)) \land (2d - y = 0 \land \neg p(d))\). After propositional reductions, we instantiate PC\(_p\) with the predicate arguments \(c\) and \(d\), due to the occurrences of the literals \(p(c)\) and \(p(d)\) in the sequent. The proof can then be closed by means of propositional rules, complementary literals, and arithmetic reasoning [9]. The final interpolant is the formula \(I = \forall x. (y - 2x \neq 0 \lor p(x))\), in which a quantifier has been introduced via ALL-LEFT-L to eliminate the constant \(d\). We prove in Appendix B.2 that no quantifier-free interpolant exists. \(\square\)

Intuitively, Theorem 4.2.3 holds because the logic PA does not provide an integer division operator. Divisibility predicates \(\alpha \mid t\) are insufficient in the presence of uninterpreted predicates, because they cannot be used within terms: no quantifier-free formula can express the statement \(\forall x. (y - 2x \neq 0 \lor p(x))\), which is equivalent to \(2 \mid y \rightarrow p(\frac{y}{2})\).

Adding integer division is sufficient to close PA+UP under interpolation. More formally, we define the logic PAID, extending PA by guarded quantified expressions

\[
\forall x. (\alpha x + t \neq 0 \lor \phi), \quad \exists x. (\alpha x + t = 0 \land \phi)
\]  

(4.2)

where \(x \in X\) ranges over variables, \(\alpha \in \mathbb{N} \setminus \{0\}\) over non-zero integers, \(t\) over terms not containing \(x\), and \(\phi\) over PAID formulae (possibly containing \(x\) as a free
variable). The logic PAID+UP is obtained by adding uninterpreted predicates to PAID. Note that the interpolant $I$ computed in Example 4.2.4 is in PAID+UP.

It is easy to extend our interpolating calculus to a sound and complete calculus for PAID+UP; the only necessary additional rules are

\[
\frac{\Gamma, [(\alpha \not\vdash t) \vee \exists x. (\alpha x + t \equiv 0 \land \phi)]_D \vdash \Delta \triangleright I}{\Gamma \vdash [\forall x. (\alpha x + t \not\equiv 0 \lor \phi)]_D, \Delta \triangleright I}
\]

\[
\frac{\Gamma \vdash [(\alpha | t) \land \forall x. (\alpha x + t \not\equiv 0 \lor \phi)]_D, \Delta \triangleright I}{\Gamma \vdash \exists x. (\alpha x + t \equiv 0 \land \phi)}
\]

with the side conditions that $\alpha \neq 0$, and that $x$ does not occur in $t$.

**Lemma 4.2.5** (Completeness). Suppose $\Gamma, \Delta$ are sets of labeled PAID+UP formulae. If the sequent $\Gamma_L, \Gamma_R \vdash \Delta_L, \Delta_R$ is valid, then there is a formula $I$ such that (i) the sequent $\Gamma \vdash \Delta \triangleright I$ is provable in the calculus of Section 4.2.1, enriched with the rules ALL-LEFT-GRD and EX.RIGHT-GRD, and (ii) $I$ is a PAID+UP formula up to normalisation of guards in expressions (4.2).

Guard normalisation is necessary in general, because interpolants generated by proofs can take the shape $\forall \bar{x}. (t_1 \not\equiv 0 \lor \cdots \lor t_k \not\equiv 0 \lor \phi)$, grouping together multiple quantifiers and guards. Such formulae can effectively be transformed into the form (4.2) (see Appendix B.3.1 for more details).

**Theorem 4.2.6.** PAID+UP is closed under interpolation.

Not all proofs in our calculus give rise to PAID+UP interpolants, however; a counterexample is given in Example 4.2.8. The following lemma provides a sufficient condition for interpolants to be PAID+UP formulae. The lemma refers to arithmetic rules of Figure 4.2.1.

**Lemma 4.2.7.** Suppose that every instantiation of the axiom (4.1) in a proof $P$ of the PAID+UP sequent $\Gamma \vdash \Delta \triangleright I$ has the form

\[
\frac{\ldots, [p(\bar{s})]_D \vdash [\bar{s} - \bar{t} \equiv 0]_F, [p(\bar{t})]_E, \ldots \triangleright J_2}{\frac{\ldots, [p(\bar{t})]_D \vdash \quad P \quad [p(\bar{t})]_E, \ldots \triangleright J_3}{\frac{\ldots, [(p(\bar{s}) \land \bar{s} - \bar{t} \equiv 0) \rightarrow p(\bar{t})]_F \vdash \quad P \quad \ldots \triangleright J_4}{\frac{\ldots, [p(\bar{s})]_D \vdash [p(\bar{t})]_E, \ldots \triangleright J_5}}}}
\]

\[
\frac{\ldots, [p(\bar{s})]_D \vdash \quad P \quad [p(\bar{t})]_E, \ldots \triangleright J_3}{\frac{\ldots, [p(\bar{t})]_D \vdash \quad P \quad [p(\bar{t})]_E, \ldots \triangleright J_3}}
\]
where \( D, E \in \{L, R \} \) and \( F \in \{D, E\} \) are arbitrary labels, and the proof \( \mathcal{Q} \) only uses the rules \textsc{red-right}, \textsc{mul-right}, \textsc{ipi-right}, \textsc{and-right-l}, and \textsc{close-eq-right} applied to an equality derived from \( \bar{s} - \bar{t} \neq 0 \). Furthermore, assume that \textsc{all-left} and \textsc{ex-right} are not applied in any other places in \( \mathcal{P} \). Then \( I \) is a \textsc{paid+up} formula up to normalisation of guards.

**Generalising the lemma.** There are various ways to relax the conditions given in the lemma. Most importantly, the restrictions on applications of axiom (4.1) are only relevant when unifying literals \( [p(\bar{s})]_D \) and \( [p(\bar{t})]_E \) with distinct labels \( D \neq E \). Applications of the axiom to literals with the same label are uncritical, because they never introduce quantifiers in interpolants. In fact, practical experience with our theorem prover \textsc{princess} shows that generated interpolants are often naturally in the \textsc{paid+up} fragment, even when no restrictions are imposed on the proof generation process.

Nevertheless, it is possible to construct proofs with interpolants outside of \textsc{paid+up}, by applying “wrong” rules in the sub-proof \( \mathcal{Q} \) of Lemma 4.2.7:

**Example 4.2.8.** Starting from \textsc{paid+up} input formulae, the following proof generates the interpolant \( \forall c. p(c) \), which is not equivalent to any \textsc{paid+up} formula:

\[
\begin{align*}
\begin{array}{c}
|p(0)|_L \vdash |p(0)|_L \triangleright \text{false} \\
|p(0)|_L, |q|_L \vdash |c = 0|_L, |q|_L \triangleright \text{false} \\
\ldots, |p(0)|_L, |q|_L, [(p(0) \land c = 0) \rightarrow p(c)]_L \vdash |c|_L, |q|_L \triangleright p(c) \\
|PC_p|_L, |PC_p|_R, |p(0)|_L, |q|_L \vdash |p(c)|_R, |q|_L \triangleright \forall c. p(c) \\
\end{array}
\end{align*}
\]

The first step in the proof is to instantiate axiom (4.1), in an attempt to unify the formula \( |p(0)|_L \) and \( |p(c)|_R \); this instantiation later introduces the unguarded quantifier \( \forall c \) in the interpolant. The proof violates the conditions in Lemma 4.2.7, because the middle sub-proof is closed using the atoms \( |q|_L \) instead of the equation \( |c \equiv 0|_L \). A correct \textsc{paid+up} interpolant for this example is false. \( \square \)

**\textsc{paid} and integer division.** Despite the presence of guarded quantifiers in \textsc{paid}, the logic is very close to simple quantifier-free assertion languages found in programming languages like Java or C. The following equivalences hold:

\[
\forall x. (\alpha x + t \neq 0 \lor \phi) \equiv (\alpha \models t) \lor [x/(t \div \alpha)]\phi, \quad (\alpha \models t) \equiv \alpha(t \div \alpha) \equiv t
\]

where \( \div \) denotes integer division. Conversely, an expression \( c \equiv t \div \alpha \) can be encoded in \textsc{paid} using axioms like \( \alpha c \leq t \land (t < \alpha c + \alpha \lor t < \alpha c - \alpha) \).
4.3 Interpolation for Uninterpreted Functions

4.3.1 A Relational Encoding of Uninterpreted Functions

For practical verification and interpolation problems, uninterpreted functions are more common and often more important than uninterpreted predicates. In the context of interpolation, functions share many properties with predicates; in particular, the quantifier-free fragment PA+UF is again not closed under interpolation, in analogy to Theorem 4.2.3.

As in the previous section, the interpolation property can be restored by means of adding integer division. To this end, we define the logic PAID+UF in a similar way to PAID, but allowing arbitrary occurrences of uninterpreted functions in terms. For reasoning and interpolation purposes, we represent functions via an encoding into uninterpreted predicates. The resulting calculus strongly resembles the congruence closure approach used in SMT solvers (e.g., [29]). To formalise the encoding, we introduce a further logic PAID+UF_p. Recall that P and F denote the vocabularies of uninterpreted predicates and functions. We assume that a fresh \((n+1)\)-ary uninterpreted predicate \(f_p \in P\) exists for every \(n\)-ary uninterpreted function \(f \in F\). The logic PAID+UF_p is then derived from PAID by incorporating occurrences of predicates \(f_p\) of the following form:

\[
\exists x. \ (f_p(t_1, \ldots, t_n, x) \land \phi) \tag{4.3}
\]

where \(x \in X\) ranges over variables, \(t_1, \ldots, t_n\) over terms that do not contain \(x\), and \(\phi\) over PAID+UF_p formulae (possibly containing \(x\)). In order to avoid universal quantifiers, we do not allow expressions (4.3) underneath negations.

Formulae in PAID+UF can uniformly be mapped to PAID+UF_p by rewriting:

\[
\phi[f(t_1, \ldots, t_n)] \rightsquigarrow \exists x. \ (f_p(t_1, \ldots, t_n, x) \land \phi[x]) \tag{4.4}
\]

provided that the terms \(t_1, \ldots, t_n\) do not contain variables bound in \(\phi\). To stay within PAID+UF_p, application of the rule underneath negations has to be avoided, which can be done by transformation to negation normal form. We write \(\phi_{RE}\) for the function-free PAID+UF_p formula derived from a PAID+UF formula \(\phi\) by exhaustive application of (4.4). Conversely, \(\phi\) can be obtained from \(\phi_{RE}\) by applying (4.4) in the opposite direction. Assuming functional consistency, the formulae \(\phi\) and \(\phi_{RE}\) are satisfiability-equivalent:
Lemma 4.3.1. Let $FC_f$ denote the functional consistency axiom:

$$FC_f = \forall \bar{x}_1, \bar{x}_2, y_1, y_2. \ ((f_p(\bar{x}_1, y_1) \land f_p(\bar{x}_2, y_2) \land \bar{x}_1 \equiv \bar{x}_2) \rightarrow y_1 \equiv y_2) \quad (4.5)$$

A PAID+UF formula $\phi$ is satisfiable exactly if $\phi_{RE} \land \bigwedge_{f \in F} FC_f$ is satisfiable.

By the lemma, it is sufficient to construct a proof of $\neg (\phi_{RE} \land \bigwedge_{f \in F} FC_f)$ in order to show that $\phi$ is unsatisfiable. The axioms $FC_f$ can be handled by ground instantiation, just like the predicate consistency axiom (4.1): whenever atoms $f_p(\bar{s}_1, t_1)$ and $f_p(\bar{s}_2, t_2)$ occur in the antecedent of a sequent, an instance of $FC_f$ can be generated using the rules \textsc{all-left-l/r} and the substitution $[\bar{x}_1/\bar{s}_1, \bar{x}_2/\bar{s}_2, y_1/t_1, y_2/t_2]$. This form of instantiation is sufficient, because predicates $f_p$ only occur in positive positions in $\phi_{RE}$, and therefore only turn up in antecedents. As before, the number of required instances can be kept under control by formulating suitable refinements [72].

4.3.2 Interpolation for PAID+UF

PAID+UF conjunctions $A \land B$ can be interpolated by constructing a proof of

$$[A_{RE}]_L, [B_{RE}]_R, \{[FC_f]_L\}_{f \in F_A}, \{[FC_f]_R\}_{f \in F_B} \vdash \emptyset \triangleright I \quad (4.6)$$

where $F_A/F_B$ are the uninterpreted functions occurring in $A/B$. Due to the soundness of the calculus, the existence of a proof guarantees that $I$ is an interpolant. Conversely, a completeness result corresponding to Lemma 4.2.5 also holds for PAID+UF$_p$. Because PAID+UF$_p$ interpolants can be translated back to PAID+UF by virtue of (4.4), we also have a closure result:

**Theorem 4.3.2.** The logic PAID+UF is closed under interpolation.

**Example 4.3.3.** We consider the PAID+UF interpolation problem $A \land B$ with

$$A = b \equiv f(2) \land f(a + 1) \equiv c \land d \equiv 1, \quad B = a \equiv 1 \land f(b) \equiv f(c) + d.$$

---

1Axiom (4.5) can also be formulated as $\forall \bar{x}_1, y_1, y_2. (f_p(x, y_1) \land f_p(\bar{x}, y_2) \rightarrow y_1 \equiv y_2)$, assuming the predicate consistency axiom (4.1). We chose (4.5) to simplify presentation.

2Note that this formulation fails if arbitrary quantifiers are allowed in $\phi$; this case would require axioms for totality of functions.
4.3. Interpolation for Uninterpreted Functions

\begin{center}
\begin{figure}
\begin{align*}
\cdots & \Rightarrow \cdots, [b \vdash x_1]_L, [x_2 \vdash c]_L, [x_1 \vdash x_2]_L \vdash [b \vdash c]_R \triangleright b \vdash c & \cdots & \Rightarrow \cdots, [y_1 \vdash y_2]_R, [d \vdash 1]_L, [y_1 \vdash y_2 + d]_R \vdash \emptyset \triangleright d \vdash 1 \\
\cdots & \Rightarrow \cdots, [f_p(b, y_1) \land f_p(c, y_2) \land b \vdash c] \Rightarrow y_1 \vdash y_2]_R \vdash \emptyset \triangleright b \vdash c \land d \vdash 1 (ii) \\
\cdots & \Rightarrow \cdots, [f_p(b, y_1)]_R, [f_p(c, y_2)]_R, [FC_f]_R, [x_1 \vdash x_2]_L \vdash \emptyset \triangleright b \vdash c \land d \vdash 1 \\
\cdots & \Rightarrow \cdots, [a \vdash 1]_R \vdash [2 \vdash a + 1]_L \triangleright a \neq 1 & \Rightarrow & \text{OR-LEFT-}^+ \triangleright I_1 (i) \\
\cdots & \Rightarrow \cdots, [f_p(2, x_1) \land f_p(a + 1, x_2) \land 2 \vdash a + 1] \Rightarrow x_1 \vdash x_2]_L \vdash \emptyset \triangleright I_1 \\
\cdots & \Rightarrow \cdots, [f_p(2, x_1)]_L, [f_p(a + 1, x_2)]_L, [FC_f]_L \vdash \emptyset \triangleright I_1 \\
\cdots & \Rightarrow \cdots, [A_{RE}]_L, [B_{RE}]_R, [FC_f]_L, [FC_f]_R \vdash \emptyset \triangleright \text{AND-LEFT}^+, \text{EX-LEFT}^+ \\
\end{align*}
\end{figure}
\end{center}

Figure 4.3: Interpolating proof of Example 4.3.3. Parts of the proof concerned with arithmetic reasoning or application of the CLOSE-* rules are not shown.

The corresponding PAID+UF<sub>p</sub> formulae are:

\[
A_{RE} = \exists x_1. \left( f_p(2, x_1) \land \exists x_2. \left( f_p(a + 1, x_2) \land b \vdash x_1 \land x_2 \vdash c \land d \vdash 1 \right) \right) \\
B_{RE} = \exists y_1. \left( f_p(b, y_1) \land \exists y_2. \left( f_p(c, y_2) \land a \vdash 1 \land y_1 \vdash y_2 + d \right) \right).
\]

The unsatisfiability of $A_{RE} \land B_{RE}$ is proven in Figure 4.3, requiring two applications of $FC_f$: (i) for the pair $f(2), f(a + 1)$, and (ii) for $f(x), f(y)$. The resulting interpolant is $I_1 = a \neq 1 \lor (b \vdash c \land d \vdash 1)$ and contains a disjunction due to splitting over an L-formula (i), and a conjunction due to (ii). \hfill \Box

As in Lemma 4.2.7, a sufficient condition for PAID+UF<sub>p</sub> interpolants can be given by restricting applications of the functional consistency axiom:

**Lemma 4.3.4.** Suppose that every instantiation of an axiom $FC_f$ in a proof $\mathcal{P}$
of (4.6) has the form

\[ \vdash \lfloor s_1 = s_2 \rfloor_F \quad \vdash \lfloor t_1 = t_2 \rfloor_F \]

where (i) \( D, E \in \{L, R\} \) and \( F \in \{D, E\} \) implies \( F = R \), (ii) \( R \in \{D, E\} \) implies \( F = R \), (iii) the proof \( \mathcal{D} \) only uses the rules RED-RIGHT, MUL-RIGHT, IPI-RIGHT, AND-RIGHT-L, and CLOSE-EQ-RIGHT applied to an equality derived from \( s_1 = s_2 \) (see Figures 3.3 and 3.4) and (iv) ALL-LEFT and EX-RIGHT are not applied in any other places in \( \mathcal{D} \). Then \( I \) is a PAID+UF formula up to normalisation of guards.

Proofs of this shape closely correspond to the reasoning of congruence closure procedures (e.g., [29]): two terms/nodes \( f(s_1) \) and \( f(s_2) \) are collapsed only once the equations \( s_1 = s_2 \) have been derived. Congruence closure can therefore be used to efficiently generate proofs satisfying the conditions of the lemma (abstracting from the additional reasoning necessary to handle the integers).

As in Section 4.2.2, it is also possible to relax the conditions of the lemma; in particular, there is no need to restrict FC\(_f\) applications with \( D = E \). The resulting interpolation procedure is very flexible, in the sense that many different interpolants can be generated from essentially the same proof. Reordering FC\(_f\) applications, for instance, changes the propositional structure of interpolants:

**Example 4.3.5.** In Example 4.3.3, the interpolant \( I_1 = a \neq 1 \lor (b \doteq c \land d \doteq 1) \) is derived using two FC\(_f\) applications (i) and (ii). Reordering the applications, so as to perform (ii) before (i), yields the interpolant \( I_2 = (a \neq 1 \lor b \doteq c) \land d \doteq 1 \).

After extracting an interpolant from a proof that contains relations encoding uninterpreted functions, the functions can be re-substituted in the interpolant:

\[ f_p(t_1, \ldots, t_n, t_0) \rightsquigarrow f(t_1, \ldots, t_n) \doteq t_0 \] (4.7)
In many practical cases (but not in general, as follows from Theorem 4.2.3), it is then possible to eliminate quantifiers from interpolants using simplification rules such as:

$$\forall x. (x - t \doteq 0 \rightarrow \phi) \rightsquigarrow [x/t]\phi$$

provided that $x$ does not occur in $t$.

### 4.3.3 Interpolation for the Theory of Extensional Arrays

The first-order theory of arrays [56] is typically encoded using uninterpreted function symbols `select` and `store` by means of the following axioms:

\[ \forall x, y, z. \ select(store(x, y, z), y) = z \quad (4.8) \]
\[ \forall x, y_1, y_2, z. \ (y_1 = y_2 \lor select(store(x, y_1, z), y_2) = select(x, y_2)) \quad (4.9) \]

Intuitively, `select(x, y)` retrieves the element of array $x$ stored at position $y$, while `store(x, y, z)` denotes the array that is identical to $x$, except that position $y$ stores value $z$. The extensional theory of arrays additionally supports equalities between arrays and is encoded using the following axiom:

\[ \forall x_1, x_2. \ (x_1 = x_2 \leftrightarrow (\forall y. select(x_1, y) = select(x_2, y))) \quad (4.10) \]

The quantifier-free theory of arrays is again not closed under interpolation, even without arithmetic, as was already noted in [58, 48]. A classical example is given by the following inconsistent formulae:

\[ A := M' = store(M, a, d) \]
\[ B := b \neq c \land select(M', b) \neq select(M, b) \land select(M', c) = select(M, c), \]

which only permit quantified interpolants, of the form

\[ \forall y_1, y_2. \ (y_1 = y_2 \lor select(M, y_1) = select(M', y_1) \lor select(M, y_2) = select(M', y_2)). \]

Naturally, combining array theory with quantifier-free Presburger arithmetic only exacerbates the problem. As we have shown in previous sections, extending PA+UP by guarded integer divisibility predicates results in a theory that is closed.

---

3In these and the following formulae, we use general equalities $s = t$ as a shorthand notation for $s - t \doteq 0$. 
under interpolation. We can extend this solution to the theory of arrays, but still only obtain closure under interpolation for small fragments of the logic (such as for formulae that do not contain the store symbol). The resulting interpolation procedure is similar in flavour to the procedures in [46, 60] and works by explicit instantiation of the array axioms. As in Section 4.2, axioms are handled lazily using the rules ALL-LEFT-L/R, which introduce quantifiers in interpolants as needed.

Array interpolation via relational encoding. To reduce array expressions to expressions involving uninterpreted predicates, we use the same relational encoding as in Section 4.3. To this end, we lift the axioms (4.8), (4.9), and (4.10) to the relational encoding as follows:

\[ AR_1 : = \forall x_1, x_2, y, z_1, z_2. \left( \text{store}_p(x_1, y, z_1, x_2) \wedge \text{select}_p(x_2, y, z_2) \rightarrow z_1 \doteq z_2 \right) \]

\[ AR_2 : = \forall x_1, x_2, y_1, y_2, z, z_1, z_2. \left( \begin{array}{l}
\text{store}_p(x_1, y_1, z, x_2) \\
\wedge \text{select}_p(x_1, y_2, z_1) \\
\wedge \text{select}_p(x_2, y_2, z_2)
\end{array} \rightarrow y_1 \doteq y_2 \lor z_1 \doteq z_2 \right) \]

\[ AR_3 : = \forall x_1, x_2. \left( \forall y, z_1, z_2. \left( \text{select}_p(x_1, y, z_1) \wedge \text{select}_p(x_2, y, z_2) \rightarrow z_1 \doteq z_2 \right) \rightarrow x_1 \doteq x_2 \right) \]

As in the previous sections, these axioms can be used in proofs by ground instantiation based on literals that occur in antecedents of sequents; in the case of AR3, it is also necessary to perform instantiation based on equations occurring in the succedent. This yields an interpolating (though incomplete) calculus for the full logic QPA+AR, and an interpolating decision procedure for the combined theory PAID+AR of Presburger arithmetic with integer division and arrays. Interpolants expressed via the relational encodings of the functions select and store can be translated into interpolants over array expressions via re-substitution rules.

Array properties. The array property fragment, introduced by Bradley et al. [8], comprises Presburger arithmetic and the theory of extensional arrays parameterised by suitable element theories. In array property formulae, integer variables may be quantified universally, provided that the matrix of the resulting quantified formula is guarded by a Boolean combination of equalities and non-strict inequalities. Using such formulae, one can express properties like equality
and sortedness of arrays, as they commonly occur in formulae extracted from
programs. Despite its expressiveness, satisfiability for this fragment was shown
to be decidable by providing an effective decision procedure [8].

Although Bradley et al. did not consider interpolation for the theory of array
properties, we observe that the decision procedure given in [8] can easily be
made interpolating using the calculus for QPA+AR provided in this chapter. The
decision procedure proceeds by reducing, in a sequence of 5 steps, array property
formulae to formulae in the combined theory of Presburger arithmetic with unin-
terpreted functions and the element theories. These 5 steps essentially correspond
to instantiation of the array axioms and of quantified parts of the input formulae,
which can be implemented using the interpolating rules provided in Figure 4.1.
The final step is a call to an interpolating decision procedure for Presburger arith-
metic and uninterpreted functions combined with suitable element theories; we
have presented such a procedure in this chapter.

We remark that the array property fragment is not subsumed by the restriction of
QPA+AR to Presburger arithmetic and array theory with guarded quantification
as allowed in PAID+UF.

4.4 Application to Software Verification

4.4.1 Chain Interpolation

To be used in the interpolation-based lazy abstraction framework described in
Section 2.3.2, a generalized version of the interpolation theorem is required that
not only guarantees the existence of single interpolants, but of chains of inter-
polants as specified in Definition 2.3.2. The existence of interpolant chains for
any unsatisfiable conjunction \( F_1 \land \cdots \land F_m \) is guaranteed by repeated application
of the standard interpolation theorem. More precisely, the (regular) interpolants
for the unsatisfiable pairs of formulae

\[
\begin{align*}
F_1 & \land (F_2 \land F_3 \land F_4 \land \cdots \land F_m) \quad \text{(interpolant } I_1) \\
(I_1 \land F_2) & \land (F_3 \land F_4 \land \cdots \land F_m) \quad \text{(interpolant } I_2) \\
(I_2 \land F_3) & \land (F_4 \land \cdots \land F_m) \quad \text{(interpolant } I_3) \\
& \quad \vdots \\
(I_{m-2} \land T_{m-1}) & \land F_m \quad \text{(interpolant } I_{m-1})
\end{align*}
\]

satisfy the chain interpolation property. Thus, interpolant chains for PA can be derived by applying \(m - 1\) times the interpolating procedure described in this chapter. Since the construction of \(m - 1\) proofs results in a significant overhead, however, the more attractive and common approach is to extract interpolant chains from a single unsatisfiability proof of \(F_1 \land \cdots \land F_m\), by considering different partitions of the conjuncts, i.e., different ways to label the formulae in the proof.

In the context of our interpolating calculus for PA, we need to show that a closed proof of

\[
\begin{align*}
[F_1]_L & \land \cdots \land [F_i]_L, [F_{i+1}]_R, [F_{i+2}]_R, \cdots, [F_m]_R \vdash \text{false} \quad \Gamma_i,
\end{align*}
\]

can be “relabeled” to a closed proof of

\[
\begin{align*}
[F_1]_L & \land \cdots \land [F_i]_L, [F_{i+1}]_L, [F_{i+2}]_R, \cdots, [F_m]_R \vdash \text{false} \quad \Gamma_{i+1},
\end{align*}
\]

such that sequent \(I_i, T_{i+1} \vdash I_{i+1}\) is valid. More generally, we show that a proof of \(\Gamma, \lfloor \Gamma' \rfloor_R \vdash \lfloor \Delta' \rfloor_R, \Delta \vdash I\) can be relabeled to proof of \(\Gamma, \lfloor \Gamma' \rfloor_L \vdash \lfloor \Delta' \rfloor_L, \Delta \vdash J\) such that \(\Gamma', I \vdash \Delta', J\) is valid.

To formalize the notion of “relabeled proof”, we need the following definitions. For two sets of labeled formulae \(\Gamma\) and \(\Gamma'\), we say that \(\Gamma'\) is a relabeled version of \(\Gamma\) iff there is a total function \(\gamma \in \Gamma \to \Gamma'\), called relabeling function, such that

(i) \(\gamma([\phi]_D) = [\phi]_E\) and
(ii) \(\gamma(t \circ 0 [\tau^A \odot 0]) = t \circ 0 [s^A \odot 0]\)

where \((\odot, \odot) \in \{(\leq, \leq), (\leq, \not=), (\not=, \leq)\}\) denotes the three possibilities to label atoms with partial interpolants and \((D, E) \in \{(L, L), (R, R), (R, L)\}\) denotes the possibility to change a \(L\)-label to a \(R\)-label. Further, we call the interpolating sequent \(\Gamma' \vdash \Delta' \vdash J\) a relabeled version of an interpolating sequent
4.4. Application to Software Verification

\[ \Gamma \vdash \Delta \triangleright I \text{ iff } \Gamma' \text{ is a relabeled version of } \Gamma \text{ and } \Delta' \text{ is a relabeled version of } \Delta. \]

Finally, we extend the notion of relabeling to proofs. Consider two closed proofs \( P' \) and \( P \) with roots \( r' \) and \( r \), respectively. We say that \( P' \) is a relabeling of \( P \), iff

(i) the interpolating sequent at root \( r' \) is a relabeled version of the interpolating sequent at root \( r \), and

(ii) the subproofs \( P'_1, \cdots, P'_n \) of \( r' \) and the subproofs \( P_1, \cdots, P_m \) of \( r \) are such that \( m = n \) and \( P'_i \) is a relabeled version of \( P_i \) for all \( i \in \{1, \cdots, m\} \).

For a given proof \( P \), we say that \( P \) can be relabeled to a proof \( P' \) iff there is a proof \( P' \) that is a relabeled version of \( P \). We are now ready to state the theorem.

**Theorem 4.4.1** (Chain interpolation). *Consider two sets \( \Gamma \) and \( \Delta \) (resp. \( \Gamma' \) and \( \Delta' \)) of labeled (resp. unlabeled) formulae. Any closed proof of the interpolating sequent \( \Gamma, [\Gamma']_R \vdash [\Delta']_R, \Delta \triangleright I \) can be relabeled to a closed proof of \( \Gamma, [\Gamma']_L \vdash [\Delta']_L, \Delta \triangleright J \) such that \( \Gamma', I \vdash \Delta', J \) is valid. (Proof pg. 116)*

### 4.4.2 Encoding of C Operations

This section discusses details of the verification of C programs using our interpolation procedures, continuing Section 2.3.2. In our experiments, the model checker WOLVERINE [80] (which uses the same infrastructure as the SATABS tool and supports a wide range of ANSI-C features) was used to process C programs. WOLVERINE repeatedly produces interpolation problems by encoding paths of the input program as conjunctions of transition relations of individual statements, formulated over the theory of bit-vector arithmetic combined with arrays. In order to pass such conjunctions to an interpolation procedure for PA with arrays, a suitable encoding of bitvector operations into unbounded linear arithmetic has to be chosen. We specify this encoding in a compact way using uninterpreted predicates and functions and axioms, similarly to the axiomatisation of the array theory in Section 4.3.3.

As a natural encoding, we consider the set \( \{-2^{n-1}, \ldots, 2^{n-1} - 1\} \subset \mathbb{Z} \) as the domain of signed bitvector arithmetic of width \( n \), and the set \( \{0, \ldots, 2^n - 1\} \subset \mathbb{Z} \) as the domain of unsigned arithmetic. To specify that some integer is a legal bitvector value, domain predicates \text{inSigned/inUnsigned} are declared that receive the bit-width \( n \) as first argument, and the integer value in question as second argument. For each C operation, a corresponding uninterpreted function is intro-
duced that receives, in addition to the operands, information such as the bit-width as explicit arguments.

As an example, we show the (somewhat simplified) definition of signed bitvector addition in the Princess input format:

--- Princess ---

```princess
//*** Declaration of uninterpreted predicates ***/
predicates { inSigned(int, int); }
//*** Declaration of uninterpreted functions ***/
functions {
  partial int shiftLeft(int, int);
  partial int addSigned(int, int, int); }

//*** Axioms ***/
forall int x, y; {shiftLeft(x, y)} {
  y > 0 -> shiftLeft(x, y) = shiftLeft(2*x, y-1))
&
forall int x; {shiftLeft(x, 0)} shiftLeft(x, 0) = x
&
forall int x, width; (inSigned(width, x) ->
  x >= -shiftLeft(1, width - 1) & x < shiftLeft(1, width - 1))
&
forall int x, y, width; {addSigned(width, x, y)} {
  (addSigned(width, x, y) = x + y |
    addSigned(width, x, y) = x + y - shiftLeft(1, width) |
    addSigned(width, x, y) = x + y + shiftLeft(1, width)) &
  inSigned(width, addSigned(width, x, y))
```

--- Princess ---

The axioms and declarations are mostly self-explanatory. As is common for SMT solvers, we specify triggers after universal quantifiers that state when and how an axiom is to be instantiated. For example, \{\text{shiftLeft}(x, y)\} states that the corresponding formula is to be instantiated whenever a term \text{shiftLeft}(s, t) occurs in a sequent. Internally, all uninterpreted functions are translated to uninterpreted predicates, and triggers are matched on the literals that occur in the antecedent of sequents.

Similar encodings can be provided for all C operations. A precise translation of non-linear operations like multiplication or bit-wise operations can be done by
case analysis over the values of their operands, which in general leads to formulae of exponential size, but is well-behaved in many cases that are practically relevant (e.g., if one of the operands is a literal).

### 4.5 Experimental Results

For our experiments, we use a development version of the model checker WOLVERINE [80], which implements the lazy abstraction approach to model checking. Our interpolation procedure is implemented in the tool iPrincess [9] and available for download.  

The following listing shows a part of an open-source C program (initialisation and shutdown of an md5 implementation) that was successfully verified not to contain any array bound violations, dereferentiation of possibly dangling pointers, or assertion violations. Comments and layout of the program were changed to accommodate the lack of space, but no modifications were made otherwise.

```c
/* nettle, low-level cryptographics library.  
 * Copyright (C) 2001 Niels Moeller  
 */

void md5init(struct md5_ctx *ctx) {
    ctx->digest[0] = 0x67452301; ctx->digest[1] = 0xefcdab89;
    ctx->digest[2] = 0x98badcfe; ctx->digest[3] = 0x10325476;
    ctx->count_l = ctx->count_h = 0; ctx->index = 0; /*1*/
}

static void md5transform(uint32_t *digest, const uint32_t *data) {
    uint32_t a, b, c, d;
    a = digest[0];
    b = digest[1];
    c = digest[2];
    d = digest[3];
    ROUND(F1, a, b, c, d, data[ 0] + 0xd76aa478, 7);
    ROUND(F1, d, a, b, c, data[ 1] + 0xe8c7b756, 12);
}
```

<sup>4</sup>http://www.philipp.ruemmer.org/iprincess.shtml
In order to verify `main` and all functions called from it, 51 program paths are extracted and handed over to the interpolation procedure, from which 519 interpolants are generated in total. The interpolants range from formulae like `select(ctx, 4) = 0` at point /*1*/ (structs are encoded as arrays) to inequalities of the form `i \leq 1` at /*2*/.
4.6 Related Work

Yorsh et al. [81] present a combination method to generate interpolants using interpolation procedures for individual theories. To be applicable, the technique requires individual theories to be equality interpolating; this is neither the case for Presburger arithmetic nor for arrays. To the best of our knowledge, it is unknown whether quantifier-free Presburger arithmetic with the integer division operator \( \div \) is equality interpolating.

Interpolation procedures for uninterpreted functions are given by McMillan [58] and Fuchs et al. [35]. The former approach uses an interpolating calculus with rules for transitivity, congruence, etc.; the latter is based on congruence closure algorithms. Our calculus in Section 4.3 has similarities with [35], but is more flexible concerning the order in which congruence rules are applied; a more systematic comparison is planned as future work. The papers [58, 35] do not consider the combination with full Presburger arithmetic.

Kapur et al. [48] present an interpolation method for arrays that works by reduction to the theory of uninterpreted functions. To some degree, the interpolation procedure of Section 4.3.3 can be considered as a lazy version of the procedure in [48], performing the reduction to uninterpreted functions only on demand.

In [46], Jhala et al. define a split prover that computes quantifier-free interpolants in a fragment of the theory of arrays, among others. The main objective of [46] is to derive interpolants in restricted languages, which makes it possible to guarantee convergence and a certain form of completeness in model checking. While our procedure is more general in that the full combined theory of PA with arrays can be handled, we consider it as important future work to integrate techniques to restrict interpolant languages into our procedure.

McMillan provides a complete procedure to generate (potentially) quantified interpolants for the full theory of arrays [60] by means of explicit array axioms. Our interpolation method resembles McMillan’s in that explicit array axioms are given to a theorem prover, but our procedure is also complete in combination with Presburger arithmetic.

Bradley et al. introduce the concept of constrained universal quantification in array theory [8], which essentially allows a single universal array index quantifier, possibly restricted to an index subrange, e.g. all indices in some range \([l,u]\). Un-
like full quantified array theory, satisfiability is decidable in Bradley’s fragment; interpolation is not considered in this work. We have discussed the relationship of this fragment to QPA+AR in Section 4.3.3.

For a discussion of related work concerning interpolation in pure quantifier-free Presburger arithmetic, we refer the reader to [9].

4.7 Summary

We have presented interpolating calculi for the theories of Presburger arithmetic combined with uninterpreted predicates (QPA+UP), uninterpreted functions (QPA+UF), and extensional arrays (QPA+AR). We have demonstrated that these extensions require the use of quantifiers in interpolants. Adding notions of guarded quantification, we therefore identified fragments of the full first-order theories that are closed under interpolation, yet are expressible in assertion languages present in standard programming languages.

As future work, we plan to extend our results to interpolating SMT solvers, particularly aiming at procedures that can be used in model checkers based on the lazy abstraction with interpolants paradigm. On the theoretical side, we plan to study the relationship between the logics discussed in this chapter, and architectures for combining interpolating procedures, e.g., [81]. We also want to investigate, possibly along the lines of [31], how our interpolation procedure for uninterpreted functions relates to existing procedures [58, 35], and whether any results can be obtained concerning the strength of computed interpolants. Finally, we plan to investigate a combination of our calculus with the Split-Prover approach in [46].
Approximation Techniques for Floating-Point Arithmetic

Floating-point arithmetic is essential for many embedded and safety-critical systems, such as in the avionics industry. Inaccuracies in floating-point calculations can cause subtle changes of the control flow, potentially leading to disastrous errors. Yet, as shown in this chapter, the verification of software containing floating-point operations can lead to hard-to-solve satisfiability instances. We present new over- and underapproximation techniques to simplify formulae containing floating-point operations. We show how to use the proposed approximations in a decision procedure based on the existing alternating abstraction framework presented in [12].

5.1 Motivation

Embedded systems are typically controlled by software that conceptually manipulates real-valued quantities, for instance measurements of environmental data. Such quantities are stored in a computer as floating-point numbers. As only few real numbers can be encoded in this format, values must generally be rounded to some nearby floating-point number.

Compared to a computation with infinite precision, rounding can influence program behavior in multiple ways. The deviation caused by rounding can lead to
unintuitive results, such as in a non-associative addition operation. Worse, the deviation can accumulate and eventually change the control flow of the program. Implementations of floating-point algorithms can be sensitive to very small variations in input. Bugs caused by such rounding errors are therefore often hard to reproduce and to test for, and have been referred to as “Heisenbugs” [36]. If undetected, they can have tragic consequences, as embedded devices are used in many mobile and ubiquitous computing environments. A prominent example is the Ariane 5 disaster, caused by an out-of-bounds 64-bit floating-point conversion. The indisputable need for reliability in embedded applications calls for precise and rigorous formal analysis methods.

Programs with floating-point arithmetic have been addressed in the past in various ways. In abstract interpretation [25], the program is (partially) executed on an abstract domain, such as real intervals. The transformations generated may, however, turn out too coarse for definite decisions on the given properties. Proof assistants are tools that prove theorems about programs (involving floating-point arithmetic) under human guidance (e.g. [6, 30]). This guide, unfortunately, must be highly skilled to direct the tool towards a proof. Both abstract interpretation and theorem proving often lack the ability to generate counterexamples for invalid properties, which is essential for debugging and for the high-impact field of automated test-vector generation.

This chapter presents a precise and sound decision procedure for (binary) floating-point arithmetic for the automatic analysis of software. The principal way of achieving this is to encode floating-point operations as functions on bit-vectors, and relying on efficient solvers for bit-vector logic, for instance those based on “bit-flattening” and subsequent SAT-solving. Unfortunately, this approach has proven to be intractable in practice, simply because it results in very large and hard-to-solve SAT instances, as we will illustrate.

A common solution to address this problem is to use over- and underapproximations of formulae (cf. Section 2.3.1). This chapter presents novel over- and underapproximating floating-point operations designed to simplify the satisfiability problem of floating-point arithmetic. We integrate the proposed approximations into a decision procedure based on alternating abstractions ([12], cf. Section 2.3.2).
5.2 Floating-Point Arithmetic

5.2.1 The IEEE Floating-Point Format

The binary floating-point format is used to represent real numbers in a computer. Specifically, the triple consisting of a sign \( s \in \{0, 1\} \), an integer-valued exponent \( e \), and a rational-valued mantissa \( m \) represents the floating-point number \((-1)^s \cdot m \cdot 2^e\). According to the IEEE standard 754, the three components are encoded using bit-vectors, resulting in the partitioned representation of a floating-point number shown in Figure 5.1.

![Figure 5.1: The three fields of an IEEE-754 floating-point number](image)

The sign bit \( s \) directly represents the sign of the floating-point number. The following two bit-vector fields are interpreted as follows:

- The bit field \( \bar{e} := e_{r-1} \ldots e_0 \) encodes the integral exponent \( e \) as a binary number.

- Together with the hidden bit, the bit field \( \bar{m} := m_{p-1} \ldots m_0 \) encodes the fractional value of the mantissa \( m \); the representation ensures that \( 0 \leq m < 2 \). The hidden bit is derived from \( \bar{e} \), and is used to distinguish normal and denormal numbers.

The widths \( r \) and \( p \) of the second and third bit fields in Figure 5.1 are called the range and the precision of the representation. The IEEE standard 754 defines two types of floating-point numbers: the single format with \((r, p) = (8, 23)\), and the double format with \((r, p) = (11, 52)\).

Unrepresentable real numbers are rounded, as we review in Section 5.2.2. Numbers that are too large are represented using the symbols \(-\infty\) and \(+\infty\). The floating-point number NaN (“Not a Number”) represents results of operations outside of real arithmetic, such as imaginary values. We call the floating-point
numbers ±∞ and NaN *special*; they are represented using reserved patterns for exponent and mantissa.

In this chapter, we manipulate floating-point formulae by varying the precision parameter \( p \), while parameter \( r \) is fixed. We denote by \( \mathbb{F}_p \) the set consisting of the floating-point numbers \((-1)^s \cdot m \cdot 2^e\) representable as a bit-vector with precision \( p \), and the special numbers ±∞.

With \( \mathbb{R}^\infty := \mathbb{R} \cup \{±\infty\} \), we have \(-\infty \leq x \leq +\infty\) for all \( x \in \mathbb{R}^\infty \). Obviously, for \( p' \leq p \), we have \( \mathbb{F}_{p'} \subseteq \mathbb{F}_p \subset \mathbb{R}^\infty \).

### 5.2.2 Floating-Point Arithmetic

The result of an operation \( a \circ b \), for \( \circ \in \{+,-,\times,\div\} \), may not be representable in \( \mathbb{F}_p \) even though \( a \) and \( b \) are in \( \mathbb{F}_p \). In such a case, an appropriate approximation is selected. For \( x \in \mathbb{R} \), define the approximations \( \lfloor x \rfloor_p \) and \( \lceil x \rceil_p \) as

\[
\lfloor x \rfloor_p := \max \{ f \in \mathbb{F}_p : f \leq x \}, \quad \text{and} \quad \lceil x \rceil_p := \min \{ f \in \mathbb{F}_p : f \geq x \}.
\]

The values \( \lfloor x \rfloor_p \) and \( \lceil x \rceil_p \) are the two floating-point numbers in \( \mathbb{F}_p \) nearest to \( x \). There is no floating-point number strictly between \( \lfloor x \rfloor_p \) and \( \lceil x \rceil_p \). If \( x \) is larger than any non-special floating-point number in \( \mathbb{F}_p \), then \( \lfloor x \rfloor_p = +\infty \); analogously for \( \lceil x \rceil_p \). The approximation values satisfy the following *nesting property*: for \( p' \leq p \), \( \lfloor x \rfloor_{p'} \leq \lfloor x \rfloor_p \leq x \leq \lceil x \rceil_p \leq \lceil x \rceil_{p'} \).

**Definition 5.2.1.** A rounding function is a function \( \text{rd}_p : \mathbb{R} \rightarrow \mathbb{F}_p \) such that, for all \( x \in \mathbb{R} \), \( \text{rd}_p(x) \in \{\lfloor x \rfloor_p, \lceil x \rceil_p\} \).

Specific rounding functions are also known as *rounding modes*. Two examples of rounding modes are *round-up* (\( \text{rd}_p = \lceil \cdot \rceil_p \)) and *round-down* (\( \text{rd}_p = \lfloor \cdot \rfloor_p \)).

The floating-point operators \( \odot_p \in \{\oplus_p, \ominus_p, \otimes_p, \oslash_p\} \) are defined as the *rounded* result of the corresponding real operators \( \circ \in \{+,-,\times,\div\} \):

\(^1\)For instance, the addition of the binary numbers \( 1.1 \cdot 2^0 \in \mathbb{F}_1 \) and \( 1.0 \cdot 2^0 \in \mathbb{F}_1 \) (1 bit fractional precision) results in \( 10.1 \cdot 2^0 \), which is not representable with 1 bit fractional precision and a mantissa \( m < 2 \).
Definition 5.2.2. For a given rounding function \( rd_p \) and an arithmetic operation \( \circ : \mathbb{R}^2 \to \mathbb{R} \), the corresponding floating-point operation \( \circ_p : \mathbb{F}_p^2 \to \mathbb{F}_p \) is defined by

\[
x \circ_p y := rd_p(x \circ y).
\]

The IEEE standard 754 extends this definition to operations with special operands, e.g. \( +\infty \oplus_p -\infty := \text{NaN} \). Note that, due to the rounding, associativity does not hold for floating-point operations, i.e., \((a \circ_p b) \circ_p c \) may differ from \(a \circ_p (b \circ_p c)\).

Floating-point arithmetic (FPA) (with precision \( p \)) is the theory defined by the structures, denoted \( (\mathbb{F}_p, \leq, \oplus_p, \ominus_p, \otimes_p, \oslash_p) \), satisfying the the definitions given above. The goal of this work is a decision procedure that determines the satisfiability of FPA formulae.

5.3 Deciding Floating-Point Arithmetic

Given a circuit implementation of an IEEE-754 compliant floating-point unit (FPU), each floating-point operation can be modeled as a formula in propositional logic, as we illustrate below. This way, a formula in FPA can — in principle — be translated to an equisatisfiable formula in propositional logic and passed to a SAT-solver to check for satisfiability. This suggests a sound and complete decision procedure for FPA. The bottleneck is of course the complexity of the resulting propositional formulae, as we demonstrate in the following. Our analysis also hints at sources for approximating these formulae in meaningful ways.

5.3.1 Addition and Subtraction

Figure 5.2 shows a high-level description of a floating-point adder/subtracter as implemented in most FPUs. An adder/subtracter is composed of three modules.

- **ALIGN.** The mantissa of the smaller operand is shifted by \(|e_a - e_b|\) bits to the right, rendering the two exponents equal.

- **ADD/SUB.** The two resulting mantissas are then added, resp. subtracted, with a standard integer adder.
5.3.2 Multiplication and Division

A high-level description of a floating-point multiplier/divider is given in Figure 5.3. Besides the rounder as described above, an FPU implements the following modules for a multiplier/divider:
5.3. Deciding Floating-Point Arithmetic

<table>
<thead>
<tr>
<th>Precision</th>
<th>ALIGN</th>
<th>ADD/SUB</th>
<th>ROUND</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>p = 5</td>
<td>295</td>
<td>168</td>
<td>572</td>
<td>1035</td>
</tr>
<tr>
<td>p = 11</td>
<td>418</td>
<td>252</td>
<td>853</td>
<td>1523</td>
</tr>
<tr>
<td>p = 17</td>
<td>561</td>
<td>336</td>
<td>1153</td>
<td>2050</td>
</tr>
<tr>
<td>p = 23</td>
<td>687</td>
<td>420</td>
<td>1447</td>
<td>2554</td>
</tr>
<tr>
<td>p = 29</td>
<td>813</td>
<td>504</td>
<td>1744</td>
<td>3061</td>
</tr>
<tr>
<td>p = 35</td>
<td>996</td>
<td>588</td>
<td>2050</td>
<td>3634</td>
</tr>
<tr>
<td>p = 41</td>
<td>1140</td>
<td>672</td>
<td>2362</td>
<td>4174</td>
</tr>
<tr>
<td>p = 47</td>
<td>1284</td>
<td>756</td>
<td>2665</td>
<td>4705</td>
</tr>
<tr>
<td>p = 52</td>
<td>1404</td>
<td>826</td>
<td>2923</td>
<td>5153</td>
</tr>
</tbody>
</table>

Table 5.1: Number of variables needed to encode a FP-adder depending on $p$

- **ADD/SUB.** The exponents of the two operands are first added, for multiplication, resp. subtracted, for division.

- **MUL/DIV.** The two mantissa are then multiplied, resp. divided.

![High-level overview of a floating-point multiplier/divider](image)

Figure 5.3: High-level overview of a floating-point multiplier/divider

Table 5.2 shows the number of propositional variables needed for a floating-point multiplier/divider, depending on the width $p$ of the mantissa. As one would expect, the multiplier/divider yields a propositional formula that is expensive to
CHAPTER 5. APPROXIMATION TECHNIQUES FOR FPA

<table>
<thead>
<tr>
<th>Precision</th>
<th>MUL/DIV</th>
<th>ADD/SUB</th>
<th>ROUND</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p = 5$</td>
<td>280</td>
<td>94</td>
<td>674</td>
<td>1048</td>
</tr>
<tr>
<td>$p = 11$</td>
<td>982</td>
<td>94</td>
<td>1287</td>
<td>2363</td>
</tr>
<tr>
<td>$p = 17$</td>
<td>2188</td>
<td>94</td>
<td>1910</td>
<td>4192</td>
</tr>
<tr>
<td>$p = 23$</td>
<td>3898</td>
<td>94</td>
<td>2258</td>
<td>6550</td>
</tr>
<tr>
<td>$p = 29$</td>
<td>6112</td>
<td>94</td>
<td>3200</td>
<td>9406</td>
</tr>
<tr>
<td>$p = 35$</td>
<td>8830</td>
<td>94</td>
<td>3855</td>
<td>12779</td>
</tr>
<tr>
<td>$p = 41$</td>
<td>12052</td>
<td>94</td>
<td>4521</td>
<td>16667</td>
</tr>
<tr>
<td>$p = 47$</td>
<td>15778</td>
<td>94</td>
<td>5193</td>
<td>21065</td>
</tr>
<tr>
<td>$p = 52$</td>
<td>19268</td>
<td>94</td>
<td>5742</td>
<td>25104</td>
</tr>
</tbody>
</table>

Table 5.2: Number of variables needed to encode a FP-multiplier depending on $p$

decide. One possibility to curb these costs is to reduce the precision $p$, as this reduces the width of the multiplier. We now discuss how formulae are approximated when $p$ is reduced.

5.4 Approximating Floating-Point Arithmetic

Reducing the precision of floating-point operations approximates the input formula: a satisfiable formula may become unsatisfiable, or vice versa. In order for the results returned by a SAT solver to remain useful, we need to be aware of the “direction” of the approximation. In this section, we discuss methods for over- and underapproximating floating-point formulae by reducing their precision.

5.4.1 Overapproximation

Reducing the precision $p$ of a floating-point operation to $p'$ causes the bits needed for the correct rounding decision to be lost, and the rounding to be based on higher-order bits. It turns out that by making the reduced-precision rounding decision nondeterministic, the reduced-precision formula overapproximates the original.
Definition 5.4.1. The open rounding operation \( \overline{rd}_{p,p'} : \mathcal{P}(\mathbb{R}) \rightarrow \mathcal{P}(\mathbb{F}_p) \) is defined as
\[
\overline{rd}_{p,p'}(X) := \left[ \lfloor X \rfloor_{p'}, \lceil X \rceil_{p'} \right] \cap \mathbb{F}_p,
\]
where \( \lfloor X \rfloor_{p'} := \min_{x \in X} \lfloor x \rfloor_{p'} \) and \( \lceil X \rceil_{p'} := \max_{x \in X} \lceil x \rceil_{p'} \).

The set \( \overline{rd}_{p,p'}(X) \) can be seen as the smallest precision-\( p \) floating-point “interval” \( Y \) such that for all \( x \in X \), the reduced-precision values \( \lfloor x \rfloor_{p'}, \lceil x \rceil_{p'} \) are in \( Y \). We use the operator \( \overline{rd}_{p,p'} \) to define corresponding open floating-point operations.

Definition 5.4.2. For an arithmetic operation \( \circ : \mathbb{R}^2 \rightarrow \mathbb{R} \), the corresponding open floating-point operation \( \overline{\circ}_{p,p'} : (\mathcal{P}(\mathbb{F}_p))^2 \rightarrow \mathcal{P}(\mathbb{F}_p) \) is defined as:
\[
X \overline{\circ}_{p,p'} Y := \overline{rd}_{p,p'}(\{ x \circ y \mid x \in X, y \in Y \}).
\]

The new floating-point operation \( \overline{\circ}_{p,p'} \) overapproximates the original operation \( \circ_p \) in the sense that \( \overline{\circ}_{p,p'} \) yields more results than \( \circ_p \), i.e., \( x \circ_p y \in \{ x \} \overline{\circ}_{p,p'} \{ y \} \), for any reduced precision \( p' \leq p \), as the following lemma shows.

Lemma 5.4.3. For \( p, p' \) with \( p' \leq p \), \( x \circ_p y \in \{ x \} \overline{\circ}_{p,p'} \{ y \} \).

Proof. By the definitions of \( \circ_p \) and \( rd_p \), \( x \circ_p y = rd_p(x \circ y) \in \{ \lfloor x \circ y \rfloor_p, \lceil x \circ y \rceil_p \} \). We estimate this set as follows:

\[
\cdots \subseteq \left[ \lfloor x \circ y \rfloor_p, \lceil x \circ y \rceil_p \right] \cap \mathbb{F}_p \quad \text{[closed interval]}
\subseteq \left[ \lfloor x \circ y \rfloor_{p'}, \lceil x \circ y \rceil_{p'} \right] \cap \mathbb{F}_p \quad \text{[nesting prop.]} \\
= \overline{rd}_{p,p'}(\{ x \circ y \}) \quad \text{[Definition 5.4.1 with X = \{ x \circ y \}]} \\
= \{ x \} \overline{\circ}_{p,p'} \{ y \} \quad \text{[Definition 5.4.2]} \]

Example 5.4.4. Consider the regular floating point addition \( 1.125 \cdot 2^{-1} \oplus_p 1.125 \cdot 2^{-2} \) which results in 1.1011 for any \( p \geq 4 \). If the precision is reduced to 3, however, the outcome has to be rounded to 1.110 since the representation of the result of the operation requires at least 4 bits of precision. In the case of the round-up mode, the result is 1.110 while it is 1.101 with the round-down mode (Table 5.3). Reducing the precision causes the bits needed for correct rounding to be lost. Thus, we need to assume that the original operation results in any floating-point
number of precision $p$ that is between $1.110$ and $1.101$. To ensure the validity of the over-approximation, we encode the floating-point operations such that they return all the floating point number of precision $p$ that are between the result that is rounded up and the result that is rounded down. This ensures that the SAT-solver can always choose the original result from the set return by the open floating-point operation. This means that the approximation is an over-approximation.

One can use the open operations to generate a formula $\overline{\phi}$ that overapproximates the original $\phi$. Each floating-point operation $\odot_p$ is replaced by an open version $\odot_{p,p'}$, for some reduced precision $p'$. The reduced precision can be chosen separately for each occurrence of a floating-point operation. We describe the generation of overapproximations next.

Let $Op(\phi)$ denote the set of floating-point operation occurrences in the FPA formula $\phi$. The reduced precision for each operation is given by a precision map, i.e., by a function $\text{precisionMap}: Op(\phi) \rightarrow \mathbb{N}$ that assigns a reduced precision to every occurrence of a floating-point operation in $\phi$. Based on the precision map, an overapproximation of $\phi$ is generated as described in Algorithm 5.1.
5.4. Approximating Floating-Point Arithmetic

Algorithm 5.1 \( \mathcal{A} \): Generate overapproximation

\textbf{Input}: \( \phi \): FPA-formula, \textit{precisionMap}: a precision map
\textbf{Output}: an overapproximation of \( \phi \)

1: \hspace{1em} \textbf{let} \( \overline{\phi} \leftarrow \phi \)
2: \hspace{1em} \textbf{for all} occurrences of floating-point operations \( \odot_p \) in \( Op(\overline{\phi}) \) \textbf{do}
3: \hspace{2em} \textbf{let} \( p' \leftarrow \text{precisionMap}(\odot_p) \)
4: \hspace{2em} \textbf{replace} \( \odot_p \) by its corresponding \( \overline{\odot}_{p,p'} \)
5: \hspace{1em} \textbf{return} \( \overline{\phi} \)

If the generated overapproximation \( \overline{\phi} \) is unsatisfiable, we conclude the unsatisfiability of the original formula \( \phi \). Otherwise, the satisfiability checker yields an assignment \( \alpha \). If the assignment also satisfies the original \( \phi \), the procedure halts and reports satisfiability. If the assignment is spurious, \( \alpha \) can be used to determine which approximated operations need a higher precision to obtain a better approximation. This is described in Algorithm 5.2.

Algorithm 5.2 \( \mathcal{R} \): Refine precision map from sat. assignment

\textbf{Input}: \textit{precisionMap}: precision map; \( \alpha \): sat. assignment
\textbf{Output}: a refined precision map for \( \phi \)

1: \hspace{1em} \textbf{let} \( \text{precisionMap}' \) be an empty map
2: \hspace{1em} \textbf{for all} operation occurrences \( \odot_p \in \phi \) \textbf{do}
3: \hspace{2em} \textbf{let} \( p' \leftarrow \text{precisionMap}(\odot_p) \)
4: \hspace{2em} \textbf{let} \( a, b \) and \( r \) be the operands and the result assigned (in \( \alpha \)) to the operation \( \overline{\odot}_{p,p'} \)
5: \hspace{2em} \textbf{if} \( r \neq a \odot_p b \) \textbf{then}
6: \hspace{3em} \textbf{let} \( p' \leftarrow p' + \text{inc} \)
7: \hspace{2em} \textbf{let} \( \text{precisionMap}'(\odot_p) \leftarrow p' \)
8: \hspace{1em} \textbf{return} \( \text{precisionMap}' \)

The algorithm refines the precision map as follows. For each operation occurrence in a term \( a \overline{\odot}_{p,p'} b = c \), where \( a, b \) and \( c \) are the constants extracted from the satisfying assignment, the algorithms checks whether \( r = a \odot_p b \). In case \( r \) is the result of the original operation \( \odot_p \) the precision is left unchanged. Otherwise, the result is spurious and we increase the precision by \( inc \) bits.
5.4.2 Underapproximation

To generate an underapproximation, we devise floating-point operations with fewer results than the original operations. Observe that if a floating-point operation with reduced precision \( p' \) yields an exact result, then the same result is obtained with the original precision \( p \). Our new floating-point operations are restricted to exact precision \( p' \) results. To formalize this idea, we define a modified rounding operator \( \text{rd}_{p,p'} \):

**Definition 5.4.5.** The no-rounding operator \( \text{rd}_{p,p'} : \mathcal{P}(\mathbb{R}) \rightarrow \mathcal{P}(\mathbb{F}_p) \) is defined as

\[
\text{rd}_{p,p'}(X) := X \cap \mathbb{F}_{p'}.
\]

The quantity \( \text{rd}_{p,p'}(\{x\}) \) equals \( \{x\} \) if no rounding is required to represent \( x \) with precision \( p' \), i.e., \( x \in \mathbb{F}_{p'} \), and the empty set otherwise, independently of \( p \). Floating-point operations \( \odot_{p,p'} \) that yield exact results only are defined in analogy to Def. 5.4.2, with \( \text{rd} \) replaced by \( \text{rd} \). These operations yield fewer results than their original counterparts \( \odot_p \). That is, if \( \{z\} \) is the result of the new exact operation \( \{x\} \odot_{p,p'} \{y\} \), then \( z \) is also the result of original operation \( x \odot_p y \):

**Lemma 5.4.6.** For \( p, p' \) with \( p' \leq p \), \( \{x\} \odot_{p,p'} \{y\} = \{z\} \) implies \( x \odot_p y = z \).

**Proof.** We have \( \{x\} \odot_{p,p'} \{y\} = \{x \odot y\} \cap \mathbb{F}_{p'} = \{z\} \), thus \( z = x \odot y \in \mathbb{F}_{p'} \subseteq \mathbb{F}_p \). From \( x \odot y \in \mathbb{F}_p \), we can conclude \( [x \odot y]_p = [x \odot y]_{p'} = x \odot y = \text{rd}_p(x \odot y) = x \odot_p y = z \).

**Example 5.4.7.** Consider Table 5.4 which illustrates the regular floating-point addition \( 1.125 \cdot 2^{-1} \oplus_3 1.000 \cdot 2^{-2} \) with a precision of 3 bits. The result is representable with 3 bits and, thus, no rounding is needed. Note that, in the case where the original precision is higher than 3, again no rounding is needed and the result is identical.

Thus, by restricting the domain of the no-rounding adder such that it is defined on operands that do not trigger rounding, we make sure that the same result is feasible with the original precision. This means that the approximation is an under-approximation.
### 5.4. Approximating Floating-Point Arithmetic

<table>
<thead>
<tr>
<th>Exp.</th>
<th>Mant.</th>
<th>Comment</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>1.001</td>
<td>Original operands</td>
<td>1.1250 · 2⁻¹</td>
</tr>
<tr>
<td>⊕₃</td>
<td>001</td>
<td>Alignment:</td>
<td>1.1250 · 2⁻¹</td>
</tr>
<tr>
<td>010</td>
<td>1.000</td>
<td>Mant. right-shifted</td>
<td>0.5000 · 2⁻²</td>
</tr>
<tr>
<td>⊕₃</td>
<td>010</td>
<td>Mant. substracted</td>
<td>1.6250 · 2⁻¹</td>
</tr>
<tr>
<td>010</td>
<td>1.101</td>
<td>Normalization</td>
<td>1.6250 · 2⁻¹</td>
</tr>
<tr>
<td>001</td>
<td>1.101</td>
<td>Result (Exact)</td>
<td>1.6250 · 2⁻¹</td>
</tr>
</tbody>
</table>

Table 5.4: A floating-point addition resulting in an exact result

**Algorithm 5.3 ⊙: Generate underapproximation**

**Input:** φ: FPA-formula, precisionMap: a precision map for φ

**Output:** an underapproximation of φ

1: let φ ← φ
2: for all occurrences of floating-point operations ⊙ₚ ∈ Op(φ) do
3: let p' ← precisionMap(⊙ₚ)
4: replace ⊙ₚ by its corresponding ⊙ₚ,p'
5: return φ

Similarly to the overapproximation case, the new operation ⊙ₚ,p' is used to generate an underapproximation of an FPA formula φ by replacing each floating-point operation with a version that is exact for some reduced precision p' as described in Algorithm 5.3. Where a satisfying assignment α is reported for the generated underapproximation φ, one can conclude that the original formula φ is also satisfiable (namely by α). If the underapproximation is unsatisfiable, however, the decision procedure returns a proof of unsatisfiability which is used to extract the unsatisfiability core of φ (i.e. the set of subformulae in φ which are used in the proof P and leading to unsatisfiability). Now it is possible to check if the constraint X ∩ Fₚ of an exact operation ⊙ₚ,p' is contained in the unsatisfiability core. If no such constraint is contained in the unsatisfiability core, we can deduce that P is also a proof for the unsatisfiability of the original formula φ. Otherwise, the unsatisfiability core can be used to identify which approximated operations need a higher precision in order to yield a more precise approximation. A detailed description of the refinement procedure is presented in Algorithm
Algorithm 5.4 $R$: Refine precision map from unsat. core

Input: precisionMap: a precision map; core: an unsatisfiability core
Output: a refined precision map for $\phi$

1: let precisionMap' be an empty map
2: for all operation occurrences $\odot_p \in Op(\phi)$ do
3: let $p' \leftarrow precisionMap(\odot_p)$
4: if the predicate $x \in F_p' \in core$ then
5: let $p' \leftarrow p' + inc$
6: let precisionMap'($\odot_p$) $\leftarrow p'$
7: return precisionMap'

5.4. The precision map is refined by the following method. For each operation $x \odot_{p,p'} y = z$ the algorithm checks whether the predicate $z \in F_p'$ from Definition 5.4.5 is in the unsatisfiability core. If it is, this suggests that the restriction to exact results in $F_p'$ was involved in the unsatisfiability of the underapproximation. Hence, we refine the precision by $inc$ bits for this operation.

Putting it all together. We can use the approximation and refinement methods presented in Sections 5.4.1 and 5.4.2 to apply the alternating approach from [12] (cf. Section 2.3.2 and Figure 2.3) to floating-point arithmetic. The abstraction procedures $\overline{A}$, $\overline{A}$ and the refinement algorithms $R$, $\overline{R}$ in the algorithm described in Figure 2.3 are described in this section.

5.5 Experimental Results

Implementation and Benchmarks. We have implemented the abstraction techniques presented in this chapter on top of the CBMC [21] model checker which is based on BMC (cf. Section 2.3.2). The benchmarks we use are derived from a variety of publicly-available C programs, selected from the SNU real-time [78] and the Mediabench benchmarks [53]. We have manually annotated our benchmarks with properties in the form of arithmetic assertions; a few of the programs already contain some (light-weight) assertions.
Table 5.5: Comparison of full flattening to alternating abstraction

**Results.** Table 5.5 compares runtimes of the full-flattening and alternating abstraction techniques. The results show that in 7 cases (out of 15) the alternating approach performs better than full flattening out of which 2 cases an order of magnitude faster. However, there are 3 instances in which the alternating abstraction is weaker than the full flattening technique. In 2 cases alternation is unable to proceed beyond the first iteration, as either the pure under- or overapproximation is already too hard. Most importantly, in half of cases in which more than one iteration (alternation) is needed, full flattening performs better than alternation. These results are due to coarse over- and underapproximations, which do not model any rounding decisions. That is, if rounding is decisive for validity of an assertion, leaving the rounding open in *every* floating-point operations results in coarse overapproximations (even with higher precision). Similarly, restricting every operation to exact results yields coarse underapproximations. As an example, consider the satisfiable FPA formula \((x \odot y) \odot z \neq x \odot (y \odot z)\). Observe that every underapproximation obtained by making the floating-point operations exact is unsatisfiable since associativity holds if no rounding occurs. On the other hand, every overapproximation is satisfiable. To obtain a satisfying assignment we rely on the ability of the decision procedure to yield an non-spurious assignment from an overapproximation. However, if *every* operation has open
rounding, spuriousness is likely. One possibility for addressing this problem is to mix approximations by overapproximating some operations and underapproximating the others. For example, the approximated formula \((x \odot y) \odot z \neq x \odot (y \odot z)\) is sufficient to find a satisfying assignment, although it is neither an over- nor an underapproximation. We develop these ideas in Chapter 6.

5.6 Related Work

A popular approach to verifying software with floating-point operations is to use proof assistants, i.e., programs that prove theorems under the guidance of a human expert. A variety of assistants have been used to prove upper bounds for the deviation from the real arithmetic result of a calculation. Examples include proofs using HOL [15], HOL-Light [43, 42, 41] and ACL2 [74, 75]. However, in case no proof is found, proof assistants typically return little evidence as to whether the proof goal was actually invalid, or whether the proof strategy was too weak to establish its validity.

An alternative approach is to use abstract interpretation [25] together with domains that can soundly approximate floating-point computations for classical static analysis. The static analyzer ASTREÉ [5] uses a number of domains which can soundly abstract floating-point computations, including intervals, octagons, polyhedra, and ellipsoid domains.

The adaptation of abstract domains for floating-point numbers is a non-trivial problem due to issues of rounding, the possibility of overflows and underflows, and division by zero errors. Relational abstract domains such as the octagon domain rely on associativity and distributivity of arithmetic operations. These properties do not hold for floating-point numbers. In ASTREÉ, floating-point expressions are therefore approximated by linear expressions over the real field with interval coefficients [64] before they are transformed into their target abstract domains. In [16], a floating-point polyhedra abstract domain based on these ideas is presented. Their linearisation technique is implemented in the APRON library for static analysis [45], which provides sound handling of floating-point expressions for a number of abstract domains.

The propagation of floating-point rounding errors has also been studied extensively in the framework of abstract interpretation [38, 39, 55, 67]. Such analyses
allow the quantification of deviations of floating-point computations from their exact result in arithmetic over the reals. Verifying floating-point programs using abstract interpretation shares with our method the advantage of being fully automatic. However, in the case where the property does not hold, it is usually difficult to obtain a counterexample that can be reproduced on the actual program.

Neither interactive theorem provers nor the use of abstract domains enjoy the characteristics of a decision procedure, namely completeness and full automation in deciding floating-point expressions. There has been some work on decision procedures for floating-point arithmetic in the field of constraint satisfaction programming (CSP). Solvers for CSP instances containing floating-point constraints have been applied to automated test-vector generation [63, 7]. This approach combines filtering the possible values of variables using interval techniques with a search procedure for finding actual floating-point values inside these intervals. Their algorithm is mainly geared towards test-vector generation, not verification. The authors state that in some cases where the calculated intervals overapproximate the concrete variable values too coarsely, their approach is unable to terminate with an answer in reasonable time. This includes the case where no such concrete solutions exist.

5.7 Summary

This chapter presents new over- and underapproximation technique for floating-point operations. These approximations are achieved by reducing the precision and modifying the rounding modes. Such modifications minimize the size of the propositional encoding needed to approximate floating point arithmetic and, thus, reduces the burden on the SAT solver. We implemented the proposed approach in the alternating abstraction framework of [12]. Experimental data suggests, however, that pure over- or underapproximations are not sufficient to yield accurate approximations where rounding and precision are decisive for the verification task.

One aspect of future work is an investigation into varying the exponent width $r$ in order to approximate a floating-point operation. While the operations on the exponent contribute only the smaller part of the propositional encoding, many programs exist that only exercise a very small range of exponent values.
Mixed Abstractions for Floating-Point Arithmetic

OVER- and underapproximation methods do not always provide effective abstractions to improve the efficiency of verification. This chapter presents a simple yet general technique for building abstractions using a mixture of over- and underapproximation transformations. We instantiate the method to a bit-accurate, sound and complete decision procedure for IEEE-compliant binary floating-point arithmetic. The procedure benefits in practice from its ability to flexibly harness both over- and underapproximations in the abstraction process. We demonstrate the potency of the procedure for the formal analysis of floating-point software.

6.1 Motivation

Classical counterexample-guided abstraction refinement (CEGAR, see Section 2.3.2) relies on overapproximations, which are refined if spurious counter-examples (in the form of spurious satisfying assignments) are encountered. In [12], a decision procedure for bit-vector arithmetic is presented that employs both types of approximations, although in a fixed alternation schedule (cf. alternating abstraction refinement in Section 2.3.2). Such approaches are too rigid for floating-point arithmetic, as some formulae do not permit effective overapprox-
imations, while others do not permit effective underapproximations.

This chapter presents a new abstraction method for checking the satisfiability of a floating-point formula $\phi$. Our algorithm permits a mixed sequence $S$ of both over- and underapproximating transformations. The formula $\psi$ resulting from applying $S$ to $\phi$ is in general neither an over- nor an underapproximation of $\phi$. Our algorithm stops whenever (i) the simplified formula $\psi$ permits a satisfying assignment that satisfies $\phi$, too, or (ii) $\psi$ is unsatisfiable and permits a resolution proof that is also a valid proof for $\phi$. If neither of the opportunities (i) and (ii) applies, $S$ needs to be refined. If $\psi$ was found to be spuriously satisfiable, the algorithm removes an overapproximating transformation from $S$, otherwise an underapproximating transformation.

The algorithm can be seen as a framework for a class of abstraction-based procedures to check the satisfiability of formulae in some logic: different choices of the transformation sequence $S$ result in different instances of our framework. For example, the work presented in [12] is an instance where $S$ strictly alternates between over- and underapproximations. CEGAR is an instance where $S$ contains only overapproximations. This chapter generalizes these methods in a way that permits a choice of approximations based on their effectiveness in simplifying the input formula. Termination of any algorithm based on our framework is guaranteed as long as the sequence $S$ can be shown to be depleted eventually.

We demonstrate the utility of the procedure on decision problems arising in bounded model checking (BMC, cf. Section 2.3.2) of ANSI-C programs.

### 6.2 The Mixed Abstraction Framework

#### 6.2.1 Overview

In Section 5.4, we have presented over- and underapproximation techniques to simplify a given floating-point formula $\phi$. Many existing procedures build either over- or underapproximations, depending on whether the goal is to show satisfiability or unsatisfiability. The two types of approximation guarantee a definite decision on the satisfiability of $\phi$ only in cases that are orthogonal for the two types. We therefore propose to combine them in a concerted effort towards analyzing $\phi$. 
To this end, we propose the abstraction framework shown in Figure 6.1, which checks the satisfiability of the input formula $\phi$. We first identify a set of eligible transformations. A transformation is a mapping that turns a FPA formula $\beta$ into a new one that over- or underapproximates $\beta$, for example by replacing some floating-point operation by its open version, as suggested in Section 5.4. The set of transformations is accordingly partitioned into subsets $Over$ and $Under$. At the beginning of the loop indicated in the figure, an implementation selects some of the eligible transformations and applies them to $\phi$ in a particular order. Note
that the resulting formula $\psi$ in general neither over- nor underapproximates $\phi$ (hence called “mixed”).

**Exit points of the loop.** Formula $\psi$ is then subject to a satisfiability check. Depending on the outcome of this check, the loop can be exited — with a definite answer — if:

(i) $\psi$ is satisfiable, and the assignment $\alpha$ returned by the solver (suitably extended) satisfies $\phi$ as well. In this case, the overall answer is “SAT”. Or:

(ii) $\psi$ is unsatisfiable, and the resolution proof $P$ returned by the solver is valid for $\phi$, too. In this case, the overall answer is “UNSAT”.

**Refinement.** If neither case (i) or (ii) applies, the approximation needs to be refined. This is done by removing some transformations from $\textit{Over}$ if $\psi$ was found to be spuriously satisfiable, otherwise from $\textit{Under}$. Which transformations to select for removal is an important implementation decision, which we discuss below in Section 6.3.2. The loop in Figure 6.1 is then reentered, and some transformations from the new sets $\textit{Over}$ and $\textit{Under}$ are applied to $\phi$.

Note that the procedure can be implemented in a both incremental and backtrackable fashion, provided the underlying SAT solver is incremental and backtrackable.

### 6.2.2 Soundness and Completeness

**Property 6.2.1.** Given a formula $\phi$, any algorithm that implements the framework in Figure 6.1, starting with finite sets $\textit{Over}$ and $\textit{Under}$ of transformations, terminates and returns “SAT” if $\phi$ is satisfiable, “UNSAT” otherwise.

*Proof.* (a) Partial correctness: The algorithm outputs “SAT” only in the case that the assignment $\alpha$ was validated successfully against the original formula $\phi$. It outputs “UNSAT” only in the case that an unsatisfiability proof for $\psi$ was found to be a valid proof of unsatisfiability for $\phi$.

(b) Termination: In each round in which the algorithm does not exit, at least one element is removed from $\textit{Over}$ or from $\textit{Under}$. When both sets are exhausted, $\psi$ and $\phi$ are equivalent, and one of the two exit conditions is trivially satisfied. \(\square\)
Note that the correctness property is independent of the distribution and scheduling of over- and underapproximations, and of the strategy for selecting elements \( o \) or \( u \) to remove in each iteration. Furthermore, it can be shown that finiteness of the sets \( \text{Over} \) and \( \text{Under} \) is not even required if one defines a suitable well-quasi-order on the set of all approximation transformations.

The Mixed Abstraction framework generalizes several other abstraction-refinement approaches to satisfiability-checking a formula. The classical CEGAR paradigm \textit{conservatively abstract–check–refine} can be seen as an instance of Figure 6.1 where the set \( \text{Under} \) is empty. Therefore, the test of whether an unsatisfiability proof for \( \psi \) is valid for \( \phi \) can be skipped in favor of an immediate answer “UNSAT”. The method presented in [12] has the property that the sequence of approximations obtained from \( \phi \) strictly alternates between strict over- and strict underapproximations. As we experimentally compare this alternating scheme to our own implementation of mixed abstraction, we sketch first how the method of [12] can be applied in a floating-point environment.

### 6.3 Applications to Floating-Point Arithmetic

#### 6.3.1 Alternating Abstractions

We can use the approximation methods presented in Section 5.4 to apply the alternating approach to floating-point arithmetic, as follows. We begin with an overapproximation \( \overline{\phi} \) of \( \phi \). To obtain \( \overline{\phi} \), the transformations in \( \text{Over} \) replace all floating-point operations \( \odot \) by \( \overline{\odot}_{p,p'} \), for some initial reduced precision \( p' \). Since \( \text{Under} \) is not applied, \( \overline{\phi} \) is an overapproximation.

If \( \overline{\phi} \) is unsatisfiable the procedure terminates. Otherwise, the decision procedure yields an assignment \( \alpha \). If \( \alpha \) also satisfies \( \phi \), the procedure halts and returns \( \alpha \) as a witness. If \( \alpha \) is spurious, we extract from it the operands \( a, b \), and the result \( r \) of each occurrence of \( \overline{\odot}_{p,p'} \). In case \( r \neq \odot_p \) we conclude that \( r \) is a spurious result and we refine (increase) the precision of \( \overline{\odot}_{p'} \); otherwise \( p' \) is left unchanged.

Next, the decision procedure builds a refined underapproximation \( \underline{\phi} \) as explained in Section 5.4.2. In this iteration, the transformations in \( \text{Under} \) replace all occurrences of \( \odot \) by \( \odot_{p,p'} \) for the refined precisions \( p' \); no transformation
from *Over* is applied. In the case where $\phi$ is satisfiable, the procedure terminates and returns $\alpha$ as an assignment for $\phi$. Otherwise, the decision procedure yields an UNSAT proof $P$ for $\phi$. If $P$ is also a proof for $\phi$, the procedure halts and returns $P$. Otherwise, it checks whether the constraint $X \cap F_p$ of an exact operation $\odot_p'$ (Definition 5.4.5) is contained in $P$. If it is, the precision is increased; otherwise it is left unchanged. The next iteration constructs a refined overapproximation. Altogether, this yields a sound and complete decision procedure alternating between over- and underapproximations.

The problem with this approach is that the alternating schedule of over- and underapproximations often leads to ineffective approximations, as some formulae are not amenable to effective overapproximations, while others do not permit effective underapproximations. As an example, consider the non-associativity formula $(a \oplus b) \oplus c \neq a \oplus (b \oplus c)$. This formula is satisfiable, as floating-point addition $\oplus$ is not associative. Satisfiability cannot be proved using an overapproximation. On the other hand, every strict underapproximation of this formula turns out to be unsatisfiable. Thus, this formula cannot be decided using *either* strict over- or strict underapproximations.

Our experimental results (Section 6.4) confirm these predictions for realistic formulae. The lesson is that an implementation of Mixed Abstraction should not be “forced” to apply either type of abstraction for pure schedule reasons. Instead, the structure of the formula itself should dictate how to approximate it. This leads to our approach of “genuinely mixed abstractions”, which we present in the following.

### 6.3.2 Genuinely Mixed Abstractions

We now detail our implementation of the mixed abstraction framework. The selection of abstractions is determined by the structure of the formula, and which approximations are most effective on it. We begin with both a very coarse overapproximation and a strong underapproximation: the result of the operation is completely nondeterministic and forced to zero at the same time. Depending on the outcome of the satisfiability check of $\psi$, either one of these approximations is refined, gradually lifting constraints or gradually increasing the precision of the operator.

The simulation of $\alpha$ on $\phi$ in the left branch in Figure 6.1 can be preceded by
a check of whether any transformation in \( \text{Over} \) was applied to \( \phi \). If not, \( \psi \) is guaranteed to be an underapproximation of \( \phi \), and “SAT” and \( \alpha \) are returned immediately. Otherwise, \( \alpha \) is suitably extended to an assignment for \( \phi \) and checked for satisfaction of \( \phi \), as suggested by the figure.

A simple and efficient pre-check whether an unsatisfiability proof \( P \) for \( \psi \) is extendable to \( \phi \) can be performed by computing the set \( \text{Var}(P) \cap \text{Var}(\text{Under}) \). This set contains the variables occurring in (the clauses of) the proof \( P \) that are involved in any of the underapproximation transformations in \( \text{Under} \). If empty, we conclude that the underapproximation transformations applied to \( \phi \) are not responsible for the unsatisfiability result for \( \psi \), which hence applies to \( \phi \), too. The emptiness test can obviously be optimized by checking first whether any transformation in \( \text{Under} \) was applied to \( \phi \).

If none of the exit tests succeeds, the algorithm selects approximation transformations \( o \) or \( u \) for removal at the end of the loop body, in order to refine the approximation. Let us look first at the case that \( \psi \) was found to be satisfiable, but the assignment \( \alpha \) is spurious (does not satisfy \( \phi \)). If there exists a transformation \( o \in \text{Over} \) such that \( \alpha \) does not satisfy the formula obtained by simplifying \( \phi \) using all transformations except \( o \), we can select such an \( o \): its removal guarantees that the spurious assignment \( \alpha \) disappears in the next round. If there is no such transformation, we select some \( o \) that affects the variables occurring in the spurious assignment \( \alpha \).

If \( \psi \) was found to be unsatisfiable, the algorithm computes the set \( \text{Var}(P) \cap \text{Var}(\text{Under}) \), as discussed above. If this set is non-empty, then there exists at least one \( u \in \text{Under} \) such that \( \text{Var}(u) \cap \text{Var}(\text{Under}) \) is non-empty. The algorithm picks such an element \( u \) for removal from \( \text{Under} \).

After refining the sets \( \text{Over} \) and \( \text{Under} \) as described, Figure 6.1 suggests to apply the refined (and smaller) sequence of approximations to \( \phi \) again. In practice, approximations selected for removal in the previous step are revoked, so that only local modifications to \( \psi \) are necessary. This allows the subsequent satisfiability check to be done incrementally, without restarts of the SAT solver.
6.4 Experimental Results

6.4.1 Implementation and Benchmarks

We have implemented the algorithm proposed in this chapter in combination with a standard bit-flattening decision procedure for integer bit-vector arithmetic. The procedure supports all operators required to model ANSI-C programs. The procedure is fully incremental: the clauses for those parts of the encoding that are not modified, and any clauses learned from those, are retained between iterations. We use MiniSAT2 [33] as our SAT solver. The performance of the integer bit-vector decision procedure we use is comparable to that of a state-of-the-art SMT-BV solver.

The benchmarks we use are derived from a variety of publicly-available C programs, selected from the SNU real-time [78] and the Mediabench benchmarks [53]; the programs we have selected make extensive use of single or double precision floating-point arithmetic. Our encoding is able to support changes of the rounding mode during the program execution, as the rounding mode may be set separately for each individual operator in the formula. However, none of the benchmark programs makes use of this feature, and we have used “round to nearest even” uniformly. We have not observed a significant impact of the specific rounding mode on the performance of either the full encoding or the abstraction-refinement procedure.

In order to obtain verification conditions, we have manually annotated our benchmarks with properties in the form of arithmetic assertions; a few of the programs already contain some (light-weight) assertions. The main property we check is the possibility of arithmetic exceptions, as defined in the IEEE floating-point arithmetic standard. These properties turn out to be difficult; an instance of a counterexample is arithmetic underflowing to zero and a subsequent division of two such numbers, which results in NaN. Such counterexamples can only be obtained if encoding artifacts such as denormal numbers are modeled in a precise manner.

After the annotation, we pass the programs to CBMC [21], which generates a total of 119 FPA decision problems. A satisfying assignment for such a formula corresponds to a counterexample that demonstrates that an assertion can be violated. Our experiments were performed on a machine running Linux on a 3 GHz
Intel Xeon CPU and with 4 MB of cache.

6.4.2 Results

Figure 6.2 compares the runtime of the mixed abstraction with the full flattening, for a timeout of 1000 s and a memory limit of 2 GB. The mixed abstraction is not dominating; in some cases (6 out of 119), the final abstraction contains almost all operators at full precision, and as a result, the immediate full flattening is faster. On the other hand, there are 58 significant instances for which the abstraction-refinement procedure terminates within the timeout, whereas the full flattening aborts due to excessive memory consumption.

Let us look at some benchmarks in more detail. For a timeout of 7 hours, Table 6.1 provides a comparison of the performance of three methods: the full flattening, an alternating abstraction as proposed in [12], and the mixed abstraction. The entries illustrate particular weaknesses and strengths of the three procedures. First of all, even small programs may result in decision problems that are simply too large (or too hard) for a full flattening. Similarly, there are instances in which the alternating abstraction is unable to proceed beyond the first iteration, as either the pure under- or overapproximation is already too hard. The mixed abstraction terminates on a number of instances that are too hard for the other procedures, but typically requires a large number of iterations. This is attributable to the extremely coarse initial abstraction, combining both over- and underapproximations. As long as the intermediate abstractions are solvable, the alternating abstraction may converge faster to a solution (e.g., consider sqrt.c, claim 2).

6.5 Related Work

Abstraction techniques are commonly considered to be the key to improve scalability of software verification. The first automatic abstraction refinement technique was developed by Chauchan et al. [52]. The technique, known as iterative abstraction refinement, first attempts to verify the property on an empty set of constraints. In case a counterexample is found, it is used to extract additional constraints to block the counterexample. This is repeated until a real counter-
example is found, or the property is proved to hold. A variation of this approach is the counterexample-guided abstraction refinement (CEGAR, cf. Section 2.3.2) approach from [20] which uses satisfiable assignments to refine overapproximations (cf. Section 2.3.2 for more details).

The approach presented in [61] uses unsatisfiability proofs in order to refine bounded model checking instances (which can be viewed as underapproximations of the unbounded model checking problem). The alternating abstraction refinement approach from [12] alternates between over- and underapproximations (cf. Section 2.3.2) to obtain a complete decision procedure for bit-vector arithmetic. Our framework can be seen as a generalization of the counterexample-guided abstraction refinement approach from [20] as well as the alternating abstraction technique presented in [12].

Figure 6.2: Runtime comparison of full flattening to mixed abstractions
6.6. **Summary**

We have presented an algorithm for iteratively approximating a complex formula by mixing both under- and overapproximations to obtain a formula $\psi$. In contrast to prior work, $\psi$ need not be an over- nor an underapproximation and can therefore be constructed in a way that yields formulae that are easy to solve. Experimental results indicate improved robustness compared to a plain flattening and to an abstraction-refinement scheme based on an alternation of over- and underapproximations.

The algorithm supports incremental solving, is complete, and produces witnesses. In particular, the ability to generate counterexamples for invalid specifications is essential not only for debugging, but also for the high-impact field of *automated test-vector generation*, where counterexamples to carefully crafted specifications translate into test cases meeting certain coverage criteria.
T HIS dissertation presents novel techniques for deriving over- and under-approximations for the verification of software modeled with theories of arithmetic, namely Presburger and floating-point arithmetic. The following section summarizes the contributions in more detail.

7.1 Summary of the Contributions

Interpolation for Presburger arithmetic and extensions thereof. The first two chapters of this thesis concentrate on the problem of deriving interpolants in the first order theory of Presburger arithmetic. We present an extension of classical sequent calculi called interpolating sequent calculi, in which rules are augmented with labels in order to extract an interpolant at the root of a closed proof. We give a set of interpolating sequent rules that form a complete interpolating calculus for the full theory of Presburger arithmetic. We extend the calculus to support uninterpreted predicates, functions and extensional arrays in order to make it practical for the the analysis of software. We show that none of these extensions support quantifier-free interpolation, even if the input formulae are quantifier-free. As interpolants are reused in the lazy abstraction with interpolants framework [59] and since quantifiers are the source of additional complexity during verification, we identify fragments with restricted forms of quantification that are closed under interpolation. In addition, to be effectively
applicable to the lazy abstraction with interpolants framework, we prove that our interpolation procedure satisfies the property of generating interpolant chains from any closed proof. Finally, we present several practical optimizations of our proposed interpolation procedure and an implementation on top of the PRINCESS theorem prover.

New approximation techniques for floating-point arithmetic. The last two chapters discuss and address the problem of deriving efficient approximations for floating-point arithmetic. We introduce over- and underapproximating floating-point operators and instrument them in the alternating abstraction framework of [12]. To further improve the efficiency of program verification in the presence of floating-point operations, we propose a new type of approximations called mixed approximations. The approach consists of identifying over- and underapproximating transformations and to apply a selection of both to a formulae, yielding an approximation that is neither a pure over- nor underapproximation. We demonstrate how to detect spuriousness using mixed abstractions and propose an abstraction refinement framework generalizing the alternating abstraction [12] and CEGAR [20] approaches. We instantiate the technique to a complete decision procedure for floating-point arithmetic and present an implementation based on the CBMC model checker.

7.2 Future Work

Quality of Presburger arithmetic interpolants for software verification. One aspect of future work in the context of interpolation for Presburger arithmetic is to empirically evaluate the quality of the generated interpolants for software verification. The presented preliminary integration of the interpolation procedure into the model checker WOLVERINE [80] requires further improvement in order to assess the quality of the generated interpolants against other techniques, such as weakest precondition computation.

Completeness of model checking by lazy abstraction based on interpolation for Presburger arithmetic. Another interesting aspect of interpolation based verification techniques is to study the conditions under which completeness is
guaranteed. Jhala et al. [46] propose to first restrict and progressively relax the interpolation language in order to derive interpolants that are more likely to establish the correctness of a program. In this way, a restricted form of completeness may be established. Future work involves studying language restriction of Presburger arithmetic and extensions thereof to improve the efficiency of software verification.

**Further applications of the mixed abstraction framework.** The last chapter of this thesis presents an approximation refinement strategy for mixed abstraction of floating-point arithmetic. A possible direction for future research is to explore further applications of the mixed abstractions to other first-order theories, such as SMT-theories. In addition, our mixed abstraction framework could be combined with techniques from abstract interpretation [25], which is a well-established technique to compute effective overapproximations.
The rules in Figure A.1 form a sound and complete calculus for PA+UP. Most of the rules are taken from [73], apart from STRENGTHEN’, which is due to [71] and replaces the OMEGA-ELIM’ rule. Slight modifications have been applied to unify notation.

**Simplification.** We denote elementary simplification steps on terms and atomic formulae in a proof with SIMP’. SIMP’ normalizes terms to the form $\alpha_1t_1 + \cdots + \alpha_nt_n$, in which $\alpha_1, \ldots, \alpha_n$ are non-zero integers and $t_1, \ldots, t_n$ are pairwise distinct variables, constants, or 1 (possibly 0 as the empty sum). Terms are put into a canonical form by sorting summands according to a well-founded ordering $<_r$.

Atomic formulae $t \neq 0, t \geq 0, t \leq 0$ are normalized by SIMP’ such that the coefficients of non-constant terms in $t$ are coprime (do not have non-trivial factors in common), and such that the leading coefficient is non-negative. This also detects that equations like $2y - 6c + 1 \neq 0$ are unsolvable and equivalent to $false$, and that an inequality like $2y - 6c + 1 \leq 0$ can be simplified and rounded to $y - 3c + 1 \leq 0$ thanks to the discreteness of the integers. All inequalities in the succedent are moved to the antecedent. A divisibility judgement $\alpha | t$ is normalized like an equation $\alpha x + t = 0$, and it is ensured that $\alpha$ and the leading coefficient of $t$ are positive.
Figure A.1: Non-interpolating rules for QPA. In EX-LEFT′ and ALL-RIGHT′, c is a constant not occurring in the conclusion. In RED′, we write φ[s] in the succedent to denote that s occurs in an arbitrary arithmetic literal in the sequent, which can also be in the antecedent. In contrast to [73], we do not allow rewriting in compound formulae. In COL-RED′, c′ is a constant that does not occur in the conclusion or in u. In DIV-LEFT′ and DIV-RIGHT′, x is an arbitrary variable and α > 0 is a literal. In FM-ELIM′ and RED′, α > 0 and β > 0 are literals.
B.1 Properties of the Interpolating Calculus

B.1.1 Soundness

Lemma 3.2.4 If an interpolating sequent $\Gamma \vdash \Delta \triangleright I$ without any PIs is provable in the calculus, then it is valid. This implies, in particular, that the sequent $\Gamma_L, \Gamma_R \vdash \Delta_L, \Delta_R$ is valid.

In order to prove the soundness of the interpolating calculus (Lemma 3.2.4), we annotate some of the rules by introducing further formulae in the premises (Figure B.1): these auxiliary formulae make it easier for us to formulate inductive properties that hold for every proof expressible in the calculus. As a convention, auxiliary formulae are written in the form $[\phi]^*_L$ and $[\phi]^*_R$. The auxiliary formulae are not necessary for completeness (it is never necessary to apply any rules to the formulae), and can therefore be completely ignored in an implementation. Similarly, soundness of the rules with auxiliary formulae directly implies the soundness of the rules without auxiliary formulae, because removing formulae from the premises could only make fewer sequents provable.

Proof of Lemma 3.2.4. Soundness is proven in multiple steps: 1. we first show that all partial interpolants occurring in a closed proof are correct. This is done by induction on the distance of a sequent from the root of the proof: assuming
that all partial interpolants in the conclusion of a rule application are correct, we prove that the partial interpolants in the rule premises are correct. 2. The validity of all sequents in a closed proof is shown by induction on the size of subproofs: assuming that all partial interpolants in the conclusion of a rule application are correct, we prove that also the conclusion is valid.

(i): Done by checking each rule. We only show some of the (interesting) cases:

- **IPI-LEFT** for the formula \( [t \doteq 0]_L \): we only need to show that the annotation of the new formula \( t \doteq 0 [t \doteq 0]_L \) is correct. By definition, we have to check that 1. the sequent \( \Gamma_L, t \doteq 0 \vdash t \doteq 0, \Delta_L \) is valid (note that the first equality comes from the labeled formula \( [t \doteq 0]_L \)), 2. the sequent \( \Gamma_R \vdash 0 \doteq 0, \Delta_R \) is valid, 3. the term \( t \) only contains constants that occur in \( \Gamma_L, t \doteq 0, \) or \( \Delta_L, \) and the term \( t - t \) only contains constants that occur in \( \Gamma_R \) or \( \Delta_R. \) All three conditions hold.

- **RED-LEFT**, for \( \circ \) being \( \leq \): suppose all partial interpolants of the conclusion are valid. This means that \( \Gamma_L \vdash t^A \doteq 0, \Delta_L \) and \( \Gamma_L \vdash s^A \doteq 0, \Delta_L \) are valid, from which we can conclude that also \( \Gamma_L \vdash s^A + \alpha \cdot t^A \doteq 0, \Delta_L \)

**Figure B.1**: Rules modified for proving soundness by adding auxiliary formulae to the premises. All other rules are as in the Figures 3.3 and 3.4 of Chapter 3.
is valid. Furthermore, $\Gamma_R \vdash t - t^A \leq 0, \Delta_R$ and $\Gamma_R \vdash s - s^A \leq 0, \Delta_R$ are valid, which implies that $\Gamma_R \vdash s - s^A + \alpha \cdot (t - t^A) \leq 0, \Delta_R$ and therefore $\Gamma_R \vdash s + \alpha \cdot t - (s^A + \alpha \cdot t^A) \leq 0, \Delta_R$ are valid. Finally, all constants of $s^A + \alpha \cdot t^A$ also occur in $s^A$ or $t^A$, and all constants of $s + \alpha \cdot t - (s^A + \alpha \cdot t^A)$ also in $s - s^A$ or $t - t^A$, so that also the vocabulary conditions are satisfied.

- **STRENGTHEN**: by assumption, the annotations of the conclusion are correct, which implies that the sequents $\Gamma_L \vdash t^A \leq 0, \Delta_L$ and $\Gamma_R \vdash t - t^A \leq 0, \Delta_R$ are valid. The correctness of the new partial interpolants is then directly guaranteed by the starred formulae in the premises.

(ii): Again, we have to check each rule. Some of the cases are:

- **CLOSE-EQ-LEFT**: suppose $t \doteq 0$ is unsatisfiable. Because $t^A \doteq 0$ is a correct partial interpolant, the sequents $\Gamma_L \vdash t^A \doteq 0, \Delta_L$ and $\Gamma_R \vdash t - t^A \doteq 0, \Delta_R$ are valid, and $t^A$ only contains constants that occur in $\Gamma_L, \Delta_L$. Given that $\Gamma_L \vdash t^A \doteq 0, \Delta_L$ is valid, so is $\Gamma_L \vdash \exists_{LA} t^A \doteq 0, \Delta_L$. On the other hand, the validity of $\Gamma_R \vdash t - t^A \doteq 0, \Delta_R$ implies that also the sequent $\Gamma_R, t^A \doteq 0 \vdash t \doteq 0, \Delta_R$ is valid, and therefore (because $t \doteq 0$ is unsatisfiable) also the sequent $\Gamma_R, t^A \doteq 0 \vdash \Delta_R$. The quantifier $\exists_{LA}$ does not concern constants in $\Gamma_R, \Delta_R$, so that also $\Gamma_R, \exists_{LA} t^A \doteq 0 \vdash \Delta_R$ is valid.

Summarizing, it has been shown that the sequents $\Gamma_L \vdash \exists_{LA} t^A \doteq 0, \Delta_L$ and $\Gamma_R, \exists_{LA} t^A \doteq 0 \vdash \Delta_R$ are valid. Because, furthermore, $\exists_{LA} t^A \doteq 0$ only contains common $L/R$-symbols, the interpolating sequent $\Gamma \vdash \Delta \triangleright \exists_{LA} t^A \doteq 0$ is valid. Because the validity condition for sequents does not mention formulae with partial interpolants, also $\Gamma, t \doteq 0 [t^A \doteq 0] \vdash \Delta \triangleright \exists_{LA} t^A \doteq 0$ is valid.

- **STRENGTHEN**: suppose each of the premises is a valid sequent, which by
definition means that the following sequents are valid:

\[
\begin{align*}
\Gamma_L, t^A \doteq 0 & \vdash E, \Delta_L, \\
\Gamma_R, t-t^A \doteq 0, E & \vdash \Delta_R, \\
\Gamma_L & \vdash I^0, \Delta_L, \\
\Gamma_R, t-t^A + 1 \lesssim 0, I^0 & \vdash \Delta_R, \\
\Gamma_L, t^A + 1 \lesssim 0 & \vdash I^1, \Delta_L, \\
\Gamma_R, I^1 & \vdash \Delta_R
\end{align*}
\] (B.1)  
\] (B.2)  
\] (B.3)  
\] (B.4)  
\] (B.5)  
\] (B.6)

Note that the arithmetic atoms in the sequents (like \( t^A \doteq 0 \) in (B.1)) stem from the \( L^*/R^* \)-formulae in the premises. Furthermore, because \( t^A \lesssim 0 \) is a correct partial interpolant, we know that the following sequents are valid:

\[
\begin{align*}
\Gamma_L & \vdash t^A \lesssim 0, \Delta_L \\
\Gamma_R & \vdash t-t^A \lesssim 0, \Delta_R
\end{align*}
\] (B.7)  
\] (B.8)

We can then construct the following proof of \( \Gamma_L \vdash I^1 \lor (E \land I^0), \Delta_L \):

\[
\begin{array}{c}
\frac{
\frac{
\frac{
\Gamma_L & \vdash t^A \doteq 0 \vdash I^1, E, \Delta_L \\
\Gamma_L & \vdash t^A + 1 \preceq 0 \vdash I^1, E, \Delta_L \\
\Gamma_L & \vdash t^A \preceq 0 \vdash I^1, E, \Delta_L
\end{array}
}{\therefore \frac{\therefore \frac{\therefore \frac{\Gamma_L \vdash t^A \preceq 0, I^1, E, \Delta_L \vdash I^1, E \land I^0, \Delta_L \vdash I^1 \lor (E \land I^0), \Delta_L}{\Gamma_L \vdash I^1 \lor (E \land I^0), \Delta_L}}{\Gamma_L \vdash I^1 \lor (E \land I^0), \Delta_L}}
\end{array}
\]
Similarly, a proof of $\Gamma_R, I^1 \lor (E \land I^0) \vdash \Delta_R$ is:

\[
\begin{align*}
(B.2) & \quad \frac{\Gamma_R, E, I^0, t - t^A \leq 0 \vdash \Delta_R}{(B.4) \quad \frac{\Gamma_R, E, I^0, t - t^A + 1 \leq 0 \vdash \Delta_R}{\Gamma_R, E, I^0, t - t^A \leq 0 \vdash \Delta_R} & \quad \frac{\Gamma_R, E, I^0 \vdash \Delta_R}{(B.6) \quad \frac{\Gamma_R, I^1 \vdash \Delta_R}{\Gamma_R, I^1 \lor (E \land I^0) \vdash \Delta_R}}
\end{align*}
\]

For the vocabulary condition, note that a constant is an $L/R$-symbol of the conclusion iff it is an $L/R$-symbol of each of the premises. This is because $t \leq 0 [t^A \leq 0]$ is annotated with a correct partial interpolant, which implies that all constants in $t$ are $L$-constants, and all constants in $t - t^A$ are $R$-constants. Therefore, the $L^*/R^*$ formulae introduced in the premises do not change vocabularies. Because each of the formulae $E, I^0, I^1$ only contains common $L/R$-constants, so does $I^1 \lor (E \land I^0)$.

This means that the sequent $\Gamma \vdash \Delta \lor I^1 \lor (E \land I^0)$ is valid, and therefore also $\Gamma, t \leq 0 [t^A \leq 0] \vdash \Delta \lor I^1 \lor (E \land I^0)$.

\[\Box\]

### B.1.2 Completeness

**Theorem 3.2.5.** Suppose $\Gamma, \Delta$ are sets of labeled formulae $|\phi|_L$ and $|\phi|_R$ such that all occurrences of existential quantifiers in $\Gamma/\Delta$ are under an even/odd number of negations, and all occurrences of universal quantifiers in $\Gamma/\Delta$ are under an odd/even number of negations. If $\Gamma_L, \Gamma_R \vdash \Delta_L, \Delta_R$ is valid, then there is a formula $I$ such that $\Gamma \vdash \Delta \lor I$ is provable.

**Proof.** The lemma follows from the completeness of the calculi in [71, 73] by means of proof lifting: given that $\Gamma_L, \Gamma_R \vdash \Delta_L, \Delta_R$ is valid, there is a proof of
this fact in the non-interpolating calculus. This proof can be lifted by replacing each rule application with an application of the corresponding interpolating rule. Completeness is proven by induction over \( n \) for the following property:

If \( \Gamma \vdash \Delta \triangleright ? \) is an interpolating sequent such that the sequent \( \Gamma' \vdash \Delta' \) (consisting of the formulae in \( \Gamma, \Delta \) without the labels) has a proof of size at most \( n \) (in the calculus from Appendix A), then there is an \( I \) such that \( \Gamma \vdash \Delta \triangleright I \) is provable in the interpolating calculus.

To prove the induction step (and base case), we have to show that each rule that can occur as the first rule application in the proof of \( \Gamma' \vdash \Delta' \) can be lifted to a rule of the interpolating calculus. We only show a few cases:

- **CLOSE-LEFT', CLOSE-RIGHT':** can be replaced with an application of CLOSE-EQ-LEFT, CLOSE-INEQ, CLOSE-EQ-RIGHT, or CLOSE-NEQ-RIGHT.

- **COL-RED':** The interpolating rules COL-RED-L/R can simulate COL-RED':

\[
\begin{align*}
\Gamma, u_L + u_R - d - d' &\vdash 0 [u_L - d \doteq 0], \ldots \vdash \Delta \\
\Gamma, u_L - d &\vdash 0 [u_L - d \doteq 0], u_R - d' &\doteq 0 [0 \doteq 0], \ldots \vdash \Delta \\
\Gamma, u_L - d &\vdash 0 [u_L - d \doteq 0], \ldots \vdash \Delta
\end{align*}
\]

The terms \( u_L, u_R \) are chosen such that \( u_L + u_R = c - u \), while at the same time \( u_L \) only contains \( L \)-constants and \( u_R \) only \( R \)-constants. The premise of the proof fragment therefore contains the formula \( c - u - d - d' \doteq 0 [u_L - d \doteq 0] \), which corresponds to the equality \( c - u - c' \doteq 0 \) in COL-RED. It is in general necessary to introduce two fresh constants \( d, d' \), in contrast to only one constant \( c' \) in COL-RED. This means that \( c' \) has to be uniformly replaced with the term \( d + d' \) in the remaining non-interpolating proof.

- **STRENGTHEN':** can be replaced with the interpolating rule STRENGTHEN. Note, that the two premises for \( t + 1 \leq 0 [t^A \doteq 0] \) and \( t + 1 \leq 0 [t^A + 1 \doteq 0] \) only differ in the partial interpolants, and therefore correspond to the same subproof of the non-interpolating proof.
B.1.3 Soundness of the \( k \)-STRENGTHEN-Rule

**Lemma 3.3.1** If the premises of the \( k \)-STRENGTHEN are valid interpolating sequents, then the interpolating sequent in the conclusion is valid. In particular this means, that \( K \) is an interpolant for the sequent \( \Gamma, t \leq 0 [t^A \leq 0] \vdash \Delta \).

**Proof.** To prove the correctness of the \( k \)-STRENGTHEN rule we will need a slightly modified version of the ordinary STRENGTHEN rule:

\[
\begin{align*}
\Gamma, t \leq 0 [t^A \leq 0], [t^A \leq 0]^*_{\ell}, [t - t^A \leq 0]^*_{r} &\vdash \Delta \parr E \\
\Gamma, t + 1 \leq 0 [t^A \leq 0], [t^A \leq 0]^*_{\ell}, [t - t^A + 1 \leq 0]^*_{r} &\vdash \Delta \parr I^0 \\
\Gamma, t + 1 \leq 0 [t^A + 1 \leq 0], [t^A + 1 \leq 0]^*_{\ell} &\vdash \Delta \parr I^1
\end{align*}
\]

The only difference between STRENGTHEN\(^\ast\) rule and the STRENGTHEN rule given in Figure B.1 is the additional \([t^A \leq 0]^*_{\ell}\) in the second premise. The correctness of the partial interpolants still follows directly from the starred formulae in the premises. Finally, note that the validity of the interpolating sequents is not affected by this change.

Let us prove the correctness of the \( k \)-STRENGTHEN rule which is as follows:

\[
\begin{align*}
\{ \Gamma, t + i \leq 0 [t^A + j \leq 0] \vdash \Delta \parr E^j_i \}_{0 \leq j \leq i < k} \\
\{ \Gamma, t + k \leq 0 [t^A + j \leq 0] \vdash \Delta \parr I^j_k \}_{0 \leq j \leq k} \\
\Gamma, t \leq 0 [t^A \leq 0] \vdash \Delta \parr K
\end{align*}
\]

The goal is to prove that the generated interpolant \( K \) of the form

\[
K := \bigvee_{0 \leq j \leq k} \left( I^j_k \land \bigwedge_{j \leq i < k} E^j_i \right)
\]

is an interpolant for the sequent \( \Gamma, t \leq 0 [t^A \leq 0] \vdash \Delta \).

We show the correctness of the \( k \)-STRENGTHEN rule for all \( k \) via two inductions. We first show that the following interpolating sequent is valid for all \( j \) and \( n \):

\[
\Gamma, t + k - n \leq 0 [t^A + j \leq 0], [t^A + j \leq 0]^*_{\ell} \vdash \Delta \parr I^j_k \land \bigwedge_{k-n \leq i < k} E^j_i \quad \text{(B.9)}
\]
For $n = 0$ this interpolating sequent reduces to the interpolating sequents generated by the $k$-STRENGTHEN rule. For the induction step, we use the assumption (B.9) as the second premise of the STRENGTHEN* rule:

\[
\begin{align*}
\Gamma, t + k - n - 1 &\vdash 0 [t_A + j \equiv 0], [t^A + j \equiv 0]_{L}^*, \ldots \vdash \Delta \vdash E_{k-n-1}^j \\
\Gamma, t + k - n &\leq 0 [t_A + j \leq 0], [t^A + j \equiv 0]_{L}^*, \ldots \vdash \Delta \vdash F \\
\Gamma, [t^A + j \equiv 0]_{L}^*, [t^A + j + 1 \leq 0]_{L}^*, \ldots &\vdash \Delta \vdash false \\
\hline
\Gamma, t + k - n - 1 &\leq 0 [t_A + j \leq 0], [t^A + j \equiv 0]_{L}^* \vdash \Delta \vdash G
\end{align*}
\]

STRENGTHEN*

By applying this rule, note that the last interpolant in the premises is trivially false since $t^A + j + 1 \leq 0$ and $t^A + j = 0$ lead to a contradiction on the left side. Using the correctness of the STRENGTHEN* rule we obtain

\[
G = false \lor (E_{k-n-1}^j \land F) \\
\equiv E_{k-n-1}^j \land I_k^j \land \bigwedge_{k-n \leq i < k} E_i^j \\
\equiv I_k^j \land \bigwedge_{k-n-1 \leq i < k} E_i^j
\]

This proves the validity of the interpolating sequent (B.9).

The second induction proof consists of proving the validity of the sequent

\[
\Gamma, t + k - n \leq 0 [t_A + j \leq 0] \vdash \Delta \vdash \bigvee_{j \leq q \leq n+j} I_k^q \land \bigwedge_{q+k-j-n \leq i < k} E_i^q \quad (B.10)
\]

In the special case where $n = k$ and $j = 0$ observe that this sequent is identical to the sequent $\Gamma, t \leq 0 [t_A \leq 0] \vdash \Delta \vdash K$ which is the main goal of this proof. For the base case $n = 0$ the sequent (B.10) reduces to the sequents generated by the $k$-STRENGTHEN rule. For the induction step, we use the assumption (B.10) as the third premise and (B.9) as the second premise of the STRENGTHEN* rule:

\[
\begin{align*}
\Gamma, t + k - n - 1 &\vdash 0 [t_A + j \equiv 0], [t^A + j \equiv 0]_{L}^*, \ldots \vdash \Delta \vdash E_{k-n-1}^j \\
\Gamma, t + k - n &\leq 0 [t_A + j \leq 0], [t^A + j \equiv 0]_{L}^*, \ldots \vdash \Delta \vdash M \\
\Gamma, t + k - n &\leq 0 [t_A + j + 1 \leq 0], [t^A + j + 1 \leq 0]_{L}^*, \ldots \vdash \Delta \vdash N \\
\hline
\Gamma, t + k - n - 1 &\leq 0 [t_A + j \leq 0] \vdash \Delta \vdash H
\end{align*}
\]

STRENGTHEN*
We show that $I_{k-n-1}^{j+1}$ is of the expected form as follows:

\[
H = N \lor (E_{k-n-1}^j \land M) \\
\equiv N \lor (E_{k-n-1}^j \land I_k^j \land \bigwedge_{k-n \leq i < k} E_i^j) \\
\equiv N \lor (I_k^j \land \bigwedge_{k-n-1 \leq i < k} E_i^j) \\
\equiv \left( \bigvee_{j+1 \leq q \leq n+j} I_k^q \land \bigwedge_{q+k-j-n-1 \leq i < k} E_i^q \right) \lor (I_k^j \land \bigwedge_{k-n-1 \leq i < k} E_i^j) \\
\equiv \bigvee_{j \leq q \leq n+1+j} I_k^q \land \bigwedge_{q+k-j-n-1 \leq i < k} E_i^q
\]

This proves the induction step and the correctness of the $k$-Strengthen rule. \qed
B.1.4 Chain interpolation property

The following recalls the definition of relabeled interpolating sequents/proofs required in this section.

For two sets of labeled formulae $\Gamma$ and $\Gamma'$, we say that $\Gamma'$ is a relabeled version of $\Gamma$ iff there is a total function $\gamma \in \Gamma \rightarrow \Gamma'$, called relabeling function, such that

(i) $\gamma([\phi]_D) = [\phi]_E$ and
(ii) $\gamma(t \circ 0[t^A \odot 0]) = t \circ 0[s^A \odot 0]$

where $(\odot, \odot), (\odot, \odot) \in \{(\models, \models), (\models, \not=), (\not=, \not=)\}$ denotes the three possibilities to label atoms with partial interpolants (cf. Definition 3.2.1) and $(D, E) \in \{(L, L), (R, R), (R, L)\}$ denotes the possibility to change a $L$-label to a $R$-label. Further, we call the interpolating sequent $\Gamma' \vdash \Delta' \Rightarrow J$ a relabeled version of an interpolating sequent $\Gamma \vdash \Delta \Rightarrow I$ iff $\Gamma'$ is a relabeled version of $\Gamma$ and $\Delta'$ is a relabeled version of $\Delta$. Finally, we extend the notion of relabeling to proofs. Consider two closed proofs $P'$ and $P$ with roots $r'$ and $r$, respectively. We say that $P'$ is a relabeling of $P$, iff

(i) the interpolating sequent at root $r'$ is a relabeled version of the interpolating sequent at root $r$, and
(ii) the subproofs $P'_1, \cdots, P'_n$ of $r'$ and the subproofs $P_1, \cdots, P_m$ of $r$ are such that $m = n$ and $P'_i$ is a relabeled version of $P_i$ for all $i \in \{1, \cdots, n\}$.

For a given proof $P$, we say that $P$ can be relabeled to a proof $P'$ iff there is a proof $P'$ that is a relabeled version of $P$.

**Theorem 4.4.1.** Consider two sets $\Gamma$ and $\Delta$ (resp. $\Gamma'$ and $\Delta'$) of labeled (resp. unlabeled) formulae. Any closed proof of $\Gamma; [\Gamma']_R \vdash [\Delta']_R, \Delta \Rightarrow I$ can be relabeled to a closed proof of $\Gamma; [\Gamma']_L \vdash [\Delta']_L, \Delta \Rightarrow J$ such that $\Gamma', I \vdash \Delta', J$ is valid.

**Proof.** The proof is organized in two steps. The first step is to show the existence of a relabeled proof for any relabeled interpolating sequent at the root of a proof $P$. We show this constructively by recursively defining a function $\eta$ that generates a relabeled proof $P'$ using the original proof $P$. In the second step, we show that any interpolating sequent in $P$ and its relabeled version in $P'$ are chain interpolating.
Since the definition of relabeled sequents/proofs does not pose any restriction on the interpolants \( \triangleright I \), we omit those in the first step of the proof.

Construction of a relabeled proof \( P' \) using \( P \). For a proof \( P \) and a relabeled sequent \( seq' \) of the root of \( P \), we construct a relabeled proof \( P' \) using a recursively defined function \( \eta : (P,seq') \mapsto P' \). Let \( seq \) be the sequent at the root of \( P \) and \( seq_1, \ldots, seq_n \) the premises of \( seq \). For any relabeled sequent \( seq'_i \) of \( seq \), we show below that it is always possible to apply a rule to \( seq'_i \) yielding premises \( seq'_i 1, \ldots, seq'_in \) such that \( seq'_i i \) is a relabeled version of \( seq_i \) for all \( i \in \{1, \ldots, n\} \).

Let \( \gamma \) and \( \delta \) be the two functions relabeling the (antecedent and succedent of the) conclusion of \( P \) to a conclusion for which we would like to build \( P' \). For \( i \in \{1, \ldots, n\} \) we show that is alway possible to construct the same number of premises such that the constructed \( i \)-th premiss in \( p' \) is a relabeling of the original \( i \)-th premiss of the form \( \Gamma, [\phi_i] R \vdash \Delta \) since the relabeling functions \( \gamma_i, \delta_i \) can be defined as

\[
\gamma_i := \gamma|_{\Gamma \setminus \{[\phi_i] R\}} \\
\gamma([\phi_i] R) := [\phi_i] D \\
\delta_i := \delta
\]  

(B.11)

The proof is similar to this case for the rules OR-LEFT-R, AND-RIGHT-L, AND-RIGHT-R, AND-LEFT, OR-RIGHT, NOT-LEFT and NOT-RIGHT.

- **OR-LEFT-R.** The conclusion is of the form \( \Gamma, [\phi_1 \lor \phi_2] R \vdash \Delta \) and, by the definition of relabeling functions, \( \gamma([\phi_1 \lor \phi_2] R) = [\phi_1 \lor \phi_2] D \) with \( D \in \{L, R\} \). Two cases arise. If \( D = L \), then the rule OR-LEFT-L is applicable to the relabeled sequent while if \( D = R \), no change of rule is required. Both cases yield the same number of premises. For \( i \in \{1, 2\} \), the generated \( i \)-th premiss is a relabeling of the original \( i \)-th premiss of the form \( \Gamma, [\phi_i] R \vdash \Delta \) since the relabeling functions \( \gamma_i, \delta_i \) can be defined as

- **IP-RIGHT-R.** The conclusion is of the form \( \Gamma \vdash [t \equiv 0] R, \Delta \) and, by the definition of relabeling functions, \( \delta([t \equiv 0] R) = [t \equiv 0] D \) with \( D \in \{L, R\} \).
Two cases arise. If $D = L$, then the rule AND-RIGHT-L is applicable to the relabeled sequent while if $D = R$, no change of rule is required. Both cases yield the same number of premises. The generated premiss is a relabeling of the original premiss $\Gamma \vdash t \equiv 0 [0 \neq 0], [t \equiv 0]_R, \Delta$ since the relabeling function $\gamma_1, \delta_1$ can be defined as

\[
\begin{align*}
\delta_1 &:= \delta \mid_{\Gamma \setminus \{t \equiv 0 [0 \neq 0]\} \cup \{[t \equiv 0]_R\}} \\
\delta_1(t \equiv 0 [0 \neq 0]) &:= \\
&= \begin{cases} 
  t \equiv 0 [t \equiv 0] & \text{if } D = L \\
  t \equiv 0 [0 \neq 0] & \text{if } D = R
\end{cases} \quad (B.12)
\end{align*}
\]

\[\gamma_1 := \gamma\]

For the rules IPI-RIGHT-L, IPI-LEFT-R and IPI-RIGHT-L, the proof is similar.

- **CLOSE-EQ-RIGHT.** The conclusion is of the form $\Gamma, t \equiv 0 [t^A \equiv 0] \vdash \Delta$ and by the definition of relabeling functions, $\gamma(t \equiv 0 [t^A \equiv 0]) = t \equiv 0 [s^A \equiv 0]$. It is therefore possible to apply the same rule yielding the same number of premises without premises. Since closure rules do not have any premises, nothing more needs to be shown for this case. The proof is similar for all other closure rules \textsc{close}-\textsc{ll}, \textsc{close}-\textsc{rr}, \textsc{close}-\textsc{lr}, \textsc{close}-\textsc{rl}, \textsc{close}-\textsc{eq}-\textsc{left}, \textsc{close}-\textsc{ineq} and \textsc{close}-\textsc{neq}-\textsc{right}.

- **COL-RED.** The conclusion is of the form $\Gamma \vdash \Delta$. Since \textsc{col-\textsc{red}} can be applied to any sequent, the rule can also by applied to any relabeled sequent $\Gamma' \vdash \Delta'$. Since $\Gamma_L$ may differ from $\Gamma'_L$, it may be necessary to choose a new term $\nu^A \neq u^A$ such that $\nu^A$ (resp. $u - \nu^A$) only contains constants from $\Gamma'_L \cup \Delta'_L$ (resp. $\Gamma'_R \cup \Delta'_R$). For a fresh constant $e$ not occurring in $\Gamma \cup \Delta$, $\Gamma' \cup \Delta'$, $u$, $u^A$ or $v^A$, we define the relabeling functions $\gamma_1, \delta_1$ as follows:

\[
\begin{align*}
\delta_1 &:= \delta \mid_{\Gamma \setminus \{t \equiv 0 [u^A \equiv 0]\}} \\
\delta_1(t - c \equiv 0 [u^A \equiv d] &:= \\
&= \begin{cases} 
  t - c \equiv 0 [\nu^A - e \equiv 0] & \text{if } \nu^A \neq u^A \\
  t - c \equiv 0 [u^A - d \equiv 0] & \text{otherwise}
\end{cases} \quad (B.13)
\end{align*}
\]

\[\gamma_1 := \gamma\]
The proof is similar for the rules RED-LEFT, RED-RIGHT, MUL-LEFT, MUL-RIGHT, DIV-LEFT and DIV-RIGHT.

- **ALL-RIGHT.** The conclusion is of the form $\Gamma \vdash \lfloor [x/c] \phi \rfloor_D, \Delta$ and, by the definition of relabeling functions, $\delta((\forall x. \phi)_D) = (\forall x. \phi)_E$ with $D, E \in \{L, R\}$. Thus, the same rule is applicable no matter the relabeling. We define the relabeling function for the premiss as follows:

$$\delta_1 := \delta|_{\Gamma \setminus \{\lfloor [x/c] \phi \rfloor_D\}}$$
$$\delta_1(\lfloor [x/c] \phi \rfloor_D) := \lfloor [x/c] \phi \rfloor_E \quad \text{(B.14)}$$

For the rule to be applicable to the relabeled conclusion, we assume that $c$ does not occur in the relabeled conclusion (this is a necessary condition for the rule to be applicable). If this assumption is not met, however, it is possible to rename the constant $c$ everywhere in the original proof to a fresh constant neither occurring in the conclusion nor in its relabeled version. This obviously does not affect the correctness of the proof.

The proof is similar for the rules EX-LEFT, ALL-LEFT-L, and ALL-LEFT-R.

- **STRENGTHEN.** The conclusion is of the form $\Gamma, \dot{t} \leq t \leq t^A \vdash \Delta$ and, by the definition of relabeling functions, $\gamma(t \leq 0[t^A \leq 0] = t \leq 0[s^A \leq 0])$. It is therefore possible to apply the same rule STRENGTHEN to any relabeled sequent obviously yielding the same number of premises. For $i \in \{1, 2, 3\}$, the generated $i$-th premiss is a relabeling of the original $i$-th premiss since, for all $i \in \{1, 2, 3\}$, the relabeling functions $\gamma, \delta_i$ can be defined as

$$\gamma := \gamma|_{\Gamma \setminus \{t \leq 0[t^A \leq 0] \cup \{t \leq 0[s^A \leq 0]\} \cup \{t \leq 0[t^A+1 \leq 0]\}}$$
$$\gamma_1(t \leq 0[t^A \leq 0]) := t \leq 0[s^A \leq 0]$$
$$\gamma_2(t \leq 0[t^A \leq 0]) := t \leq 0[s^A \leq 0]$$
$$\gamma_3(t \leq 0[t^A+1 \leq 0]) := t \leq 0[s^A+1 \leq 0]$$
$$\delta_i := \delta \quad \text{(B.15)}$$

For the rules FM-ELIM, SPLIT-EQ and SPLIT-NEQ, the proof follows the same lines.
This shows the construction of a relabeled proof for any relabeled interpolating sequent seq' of the interpolating sequent at the root of \( P \). In the following, let \( P' \) denote the relabeled proof generated by \( \eta(P, seq') \), i.e., \( P' := \eta(P, seq') \).

**Chain interpolation property in \( P \) and \( P' \).** The second step is to inductively prove the chain interpolation property. To this end, we need the following definition. Given a set of labeled formulae \( \Gamma \) and a relabeling function \( \gamma \in \Gamma \rightarrow \Gamma' \) to a set of labeled formulae \( \Gamma' \), we define the unary operator \( \Lambda_{\gamma} \) as

\[
\Lambda_{\gamma}(\Gamma) := \{ \phi \mid \gamma(\lceil \phi \rceil_{R}) = \lceil \phi \rceil_{L}\} \cup \\
\{ s^{A} - t^{A} \odot 0 \mid \gamma(t \circ 0[s^{A} \odot 0]) = t \circ 0[t^{A} \odot 0] \text{ and } s^{A} \neq t^{A}\}
\]

where \( (\circ, \odot), (\odot, \odot) \in \{ (\not=, \not=), (\not=, \not=), (\leq, \leq) \} \). This definition captures the difference between the contribution of \( A \) in \( \Gamma \) and the contribution of \( A \) in \( \Gamma' \) (including the contribution of the partial interpolants). Note that the \( \Lambda_{\gamma} \) operator obviously satisfies the additivity property:

\[
\Lambda_{\gamma}(\Gamma_{1} \cup \Gamma_{2}) = \Lambda_{\gamma}(\Gamma_{1}) \cup \Lambda_{\gamma}(\Gamma_{2}) \quad \text{(additivity property)}
\]

We say that an interpolating sequent \( \Gamma \vdash \Delta \rhd \Gamma' \) and a relabeled interpolating sequent with interpolants \( J \) are **chain interpolating** iff the sequent \( \Lambda_{\gamma}(\Gamma), I \vdash \Lambda_{\delta}(\Delta), J \) is valid.

We inductively show that any interpolating sequent in \( P \) and its relabeled version in \( P' \) are chain interpolating. The induction hypothesis is to assume that the property pairwise holds for the premises of the roots of \( P \) and \( P' \), while the induction step is to show that the property is preserved for the conclusions (roots) of \( P \) and \( P' \) (remember that the interpolants are propagated back to the root).

More precisely, for \( i \in \{1, \cdots, n\} \), we assume that the \( i \)-th premiss and its relabeled version with interpolant \( J \) are chain interpolating (with relabeling functions \( \gamma_{i}, \delta_{i} \) as defined in the first part of the proof). Depending on which rule derives the premisses, we show that the conclusion and its relabeled version are also chain interpolating:

- **OR-LEFT-R.** For \( i \in \{1, 2\} \), we assume the validity of the sequent \( \Lambda_{\gamma_{i}}(\Gamma \cup \{ \lceil \phi_{i} \rceil_{R}\}), I_{i} \vdash \Lambda_{\delta_{i}}(\Delta), I'_{i} \). As defined in the first step of the proof
B.1. Properties of the Interpolating Calculus

(Equations (B.11)), $\gamma|_\Gamma = \gamma|_\Gamma$ and $\delta = \delta$. Thus, we deduce that

\[
\Lambda_\gamma(\Gamma \cup \{[\phi_i]_R\}) = \Lambda_\gamma(\Gamma) \cup \Lambda_\gamma(\{[\phi_i]_R\}) \quad \text{additivity property}
\]

\[
= \Lambda_\gamma(\Gamma) \cup \Lambda_\gamma(\{[\phi_i]_R\}) \quad \text{since } \gamma|_\Gamma = \gamma|_\Gamma \quad \text{(B.16)}
\]

\[
\Lambda_{\delta_i}(\Delta) = \Lambda_\delta(\Delta) \quad \text{since } \delta_i = \delta \quad \text{(B.17)}
\]

From the first step of the proof (Equations (B.11)), we also know that $\gamma([\phi_i]_R) = [\phi_i]_D$ and $\gamma([\phi_1 \lor \phi_2]_R) = [\phi_1 \lor \phi_2]_D$ with $D \in \{L, R\}$. Thus, if $D = L$ we obtain

\[
\Lambda_\gamma(\{[\phi_i]_R\}) = \{\phi_i\} \quad \text{(B.18)}
\]

\[
\Lambda_\gamma(\{[\phi_1 \lor \phi_2]_R\}) = \{\phi_1 \lor \phi_2\} \quad \text{(B.19)}
\]

Using (B.16), (B.17) and (B.18), we conclude that the interpolating sequents $\Lambda_\gamma(\Gamma), \phi_1, I_1 \vdash \Lambda_\delta(\Delta), I'_1$ and $\Lambda_\gamma(\Gamma), \phi_2, I_2 \vdash \Lambda_\delta(\Delta), I'_2$ are valid. Because of the fact that two valid implications $\mu \rightarrow \nu$ and $\mu' \rightarrow \nu'$ entail the validity of $\mu \lor \mu' \rightarrow \nu \lor \nu'$ we deduce the validity of

\[
\Lambda_\gamma(\Gamma), \phi_1 \lor \phi_2, I_1 \land I_2 \vdash \Lambda_\delta(\Delta), I'_1 \lor I'_2 \quad \text{and}
\]

\[
\Lambda_\gamma(\Gamma \cup \{[\phi_1 \lor \phi_2]_R\}), I_1 \land I_2 \vdash \Lambda_\delta(\Delta), I'_1 \lor I'_2 \quad \text{(B.20)}
\]

The transition to (B.20) is achieved using the additivity property and the Equation (B.19). Similarly, if $D = R$ we know that $\Lambda_\gamma(\{[\phi_i]_R\}) = \emptyset$ and $\Lambda_\gamma(\{[\phi_1 \lor \phi_2]_R\}) = \emptyset$ which means that the sequents $\Lambda_\gamma(\Gamma), I_1 \vdash \Lambda_\delta(\Delta), I'_1$ and $\Lambda_\gamma(\Gamma), I_1 \vdash \Lambda_\delta(\Delta), I'_2$ are valid, thus showing the validity of (B.20).

The proof is similar to this case for the rules OR-LEFT-L, AND-RIGHT-R, AND-RIGHT-L, AND-LEFT, OR-RIGHT, NOT-LEFT and NOT-RIGHT.

- **IPI-RIGHT-R.** We assume the validity of the sequent $\Lambda_\gamma(\Gamma), I_1 \vdash \Lambda_\delta(\Delta \cup \{t \vdash 0 [0 \neq 0], [t \vdash 0]_R\}), I'_1$. From the first step of the proof (Equations (B.12)) we know that, if $D = L$ then $\delta_1(t \vdash 0 [0 \neq 0]) = t \vdash 0$ and $\delta([t \vdash 0]_R) = [t \vdash 0]_L$. Thus:

\[
\Lambda_{\delta_1}(\{t \vdash 0 [0 \neq 0]\}) = \{t - 0 \vdash 0\} \quad \text{(B.21)}
\]

\[
\Lambda_\delta(\{[t \vdash 0]_R\}) = \{t \vdash 0\} \quad \text{(B.22)}
\]
Similarly to the OR-LEFT-R case, we use (B.21), (B.22), the additivity property and the fact that $\gamma_1 = \gamma$ and $\delta_1|_{\Gamma \cup \{t \vdash 0\}_R} = \delta|_{\Gamma \cup \{t \vdash 0\}_R}$ (Equations (B.12)) to deduce that

$$\Lambda_{\gamma}(\Gamma), I_1 \vdash I'_1, \Lambda_{\delta}(\Delta \cup \{t \vdash 0\}) \quad \text{(B.23)}$$

holds. In the case where $D = R$, we know that $\Lambda_{\delta_1}(t \vdash 0[0 \not\vdash 0]) = \emptyset$ $\Lambda_{\delta}(\{t \vdash 0\}_R) = \emptyset$ and we similarly deduce (B.23).

For the rules IPI-RIGHT-L, IPI-LEFT-R and IPI-RIGHT-L, the proof is similar.

- **CLOSE-EQ-RIGHT.** By definition of the relabeling function $\gamma$, we know that $\gamma(0 \vdash 0[t^A \vdash 0]) = 0 \vdash 0[s^A \vdash 0]$. We directly show that the sequent

$$\Lambda_{\gamma}(\Gamma \cup \{0 \vdash 0[t^A \vdash 0]\}), \exists_{LAT} t^A \not\vdash 0 \vdash \Lambda_{\delta}(\Delta), \exists_{LAS} s^A \not\vdash 0 \quad \text{(B.24)}$$

is valid. Due to the additivity property, showing the validity of (B.24) is equivalent to showing the validity of

$$\Lambda_{\gamma}(\Gamma), \Lambda_{\gamma}(\{0 \vdash 0[t^A \vdash 0]\}), \exists_{LAT} t^A \not\vdash 0 \vdash \Lambda_{\delta}(\Delta), \exists_{LAS} s^A \not\vdash 0$$

By the definition of $\Lambda_{\gamma}$ we know that if $s^A \not\equiv t^A$ then $\Lambda_{\gamma}(\{0 \vdash 0[t^A \equiv 0]\}) = \{t^A - s^A \equiv 0\}$ and $\Lambda_{\gamma}(\{0 \vdash 0[t^A \equiv 0]\}) = \emptyset$ otherwise. In the case where $s^A \not\equiv t^A$ we obtain the sequent

$$\Lambda_{\gamma}(\Gamma), t^A - s^A \equiv 0, \exists_{LAT} t^A \not\vdash 0 \vdash \Lambda_{\delta}(\Delta), \exists_{LAS} s^A \not\vdash 0$$

and for the case where $s^A = t^A$ we obtain the sequent

$$\Lambda_{\gamma}(\Gamma), \exists_{LAT} t^A \not\vdash 0 \vdash \Lambda_{\delta}(\Delta), \exists_{LAT} t^A \not\vdash 0$$

For both cases the two sequents are obviously valid which proves that (B.24) is valid.

The proof is similar for all other closure rules CLOSE-LL, CLOSE-RR, CLOSE-LR, CLOSE-RL, CLOSE-EQ-LEFT, CLOSE-INEQ and CLOSE-NEQ-RIGHT.

- **COL-RED.** We assume that the sequent

$$\Lambda_{\gamma_1}(\Gamma \cup \{u - c \vdash 0[u^A - d \vdash 0]\}), I_1 \vdash \Lambda_{\delta_1}(\Delta), I'_1 \quad \text{(B.25)}$$
is valid. From the definitions of $\gamma_1$, $\delta_1$ in the first part of the proof (Equations (B.13)) we deduce that, if $u^A \neq v^A$, then

$$\Lambda_{\gamma} (\{ u - c \equiv 0 [u^A - d \equiv 0] \}) = \{ u^A - d - (v^A - e) \equiv 0 \}$$  \hspace{1cm} (B.25)

Using (B.25), the additivity property and that $\gamma_1 \mid \Gamma = \gamma \mid \Gamma$ as well as $\delta_1 = \delta$ we conclude that

$$\Lambda_{\gamma} (\Gamma), u^A - d - (v^A - e) \equiv 0, I_1 \vdash \Lambda_{\delta} (\Delta), I'_1$$  \hspace{1cm} (B.26)

is valid. Due to the validity of (B.26), we know that the implication

$$\Lambda_{\gamma} (\Gamma) \land u^A - d - (v^A - e) \equiv 0 \land I_1 \rightarrow \Lambda_{\delta} (\Delta) \lor I'_1$$  \hspace{1cm} (B.27)

is valid. Further, since $e$ is a fresh constant not occurring in $\Gamma \cup \Delta$, $\Gamma' \cup \Delta'$, $u$, $u^A$ or $v^A$, we know that $e$ does not occur in (B.26) and (B.27) either (the $\Lambda$ operator does not introduce any constants). In particular, (B.27) has to hold for any constant $e$ which entails the validity of the implication

$$\Lambda_{\gamma} (\Gamma) \land I_1 \rightarrow \Lambda_{\delta} (\Delta) \lor I'_1$$  \hspace{1cm} (B.28)

Finally, this means that the sequent

$$\Lambda_{\gamma} (\Gamma), I_1 \vdash \Lambda_{\delta} (\Delta), I'_1$$  \hspace{1cm} (B.29)

is valid. The case where $u^A \neq v^A$ is simpler. From the first part of the proof (Equations (B.13)) we know that

$$\Lambda_{\gamma} (\{ u - c \equiv 0 [u^A - d \equiv 0] \}) = \emptyset$$

and we directly deduce the validity of (B.29).

The proof is similar for the rules RED-LEFT, RED-RIGHT, MUL-LEFT, MUL-RIGHT, DIV-LEFT and DIV-RIGHT.

- **ALL-RIGHT.** We assume that the sequent

$$\Lambda_{\gamma} (\Gamma), I_1 \vdash \Lambda_{\delta_1} (\Delta \cup \{ [[x/c] \phi]_D \}), I'_1$$

is valid. From the definitions of $\gamma_1$, $\delta_1$ in the first part of the proof we deduce that, if $D \neq E$, then

$$\Lambda_{\delta} (\{ [[x/c] \phi]_D \}) = \{ [x/c] \phi \}$$  \hspace{1cm} (B.30)
Using (B.30), the additivity property and that \( \delta_{1|\Delta} = \delta|\Delta \) as well as \( \gamma_1 = \gamma \) we conclude that the sequent

\[
\Lambda_{\gamma}(\Gamma), I_1 \vdash \Lambda_{\delta}(\Delta), \lfloor x/c \rfloor \phi, I'_1
\]

(B.31)

is valid, which means that the implication

\[
\Lambda_{\gamma}(\Gamma) \land I_1 \rightarrow \Lambda_{\delta}(\Delta) \lor \lfloor x/c \rfloor \phi \lor I'_1
\]

(B.32)

is valid. Further, since \( c \) is a fresh constant neither occurring in the conclusion nor its relabeled version (see first part of the proof) and since the \( \Lambda \) operator does not introduce any new constants, we conclude that the implication

\[
\Lambda_{\gamma}(\Gamma) \land I_1 \rightarrow \Lambda_{\delta}(\Delta) \lor \forall x. \phi \lor I'_1
\]

(B.33)

is valid. Finally, this entails that the sequents

\[
\Lambda_{\gamma}(\Gamma), I_1 \vdash \Lambda_{\delta}(\Delta), \forall x. \phi, I'_1
\]

(B.34)

are valid. The transition to (B.34) is achieved using the additivity property and the fact that \( \Lambda_{\delta}(\lfloor \forall x. \phi \rfloor_D) = \{\forall x. \phi\} \) from the first part of the proof.

In the case where \( D = E \), we know that \( \Lambda_{\delta_1}(\lfloor [x/c] \phi \rfloor_E) = 0 \) and that \( \Lambda_{\delta}(\lfloor \forall x. \phi \rfloor_E) = 0 \) and we similarly deduce the validity of (B.34).

The proof is similar for the rules EX-LEFT, ALL-LEFT-L, and ALL-LEFT-R.

- **STRENGTHEN.** We assume that the sequents

\[
\Lambda_{\gamma_1}(\Gamma \cup \{t \equiv 0[r^A \equiv 0]\}), I_1 \vdash \Lambda_{\delta_1}(\Delta), I'_1
\]

\[
\Lambda_{\gamma_2}(\Gamma \cup \{t \leq 0[r^A \leq 0]\}), I_2 \vdash \Lambda_{\delta_2}(\Delta), I'_2
\]

\[
\Lambda_{\gamma_3}(\Gamma \cup \{t \leq 0[r^A + 1 \leq 0]\}), I_3 \vdash \Lambda_{\delta_3}(\Delta), I'_3
\]

are valid. From the definitions of \( \gamma_i, \delta_i \) in the first part of the proof (Equations (B.15)) we deduce that, if \( s^A \neq r^A \), then

\[
\Lambda_{\gamma_1}(t \equiv 0[r^A \equiv 0]) = \{s^A - r^A \equiv 0\}
\]

(B.35)

\[
\Lambda_{\gamma_2}(t \leq 0[r^A \leq 0]) = \{s^A - r^A \leq 0\}
\]

(B.36)

\[
\Lambda_{\gamma_3}(t \leq 0[r^A + 1 \leq 0]) = \{s^A + 1 - (r^A + 1) \leq 0\}
\]

(B.37)

\[
\Lambda_{\gamma}(t \leq 0[r^A \leq 0]) = \{s^A - r^A \leq 0\}
\]

(B.38)
Using (B.35)-(B.38), the additivity property and that $\gamma|_{\Gamma} = \gamma|_{\Gamma}$ as well as $\delta_i = \delta$ (Equations (B.15)) we conclude that the sequents

$$\Lambda_\gamma(\Gamma), s^A - t^A \equiv 0, I_1 \vdash \Lambda_\delta(\Delta), I'_1 \quad \text{(B.39)}$$

$$\Lambda_\gamma(\Gamma), s^A - t^A \leq 0, I_2 \vdash \Lambda_\delta(\Delta), I'_2 \quad \text{(B.40)}$$

$$\Lambda_\gamma(\Gamma), s^A - t^A \leq 0, I_3 \vdash \Lambda_\delta(\Delta), I'_3 \quad \text{(B.41)}$$

are valid. Using (B.39) and (B.40) and because of the fact that two valid implications $\mu \rightarrow \nu$ and $\mu' \rightarrow \nu'$ entail the validity of $\mu \land \mu' \rightarrow \nu \land \nu'$ we deduce the validity of the validity of

$$\Lambda_\gamma(\Gamma), s^A - t^A \leq 0, (I_1 \land I_2) \vdash \Lambda_\delta(\Delta), I'_1 \land I'_2$$

and

$$\Lambda_\gamma(\Gamma), s^A - t^A \leq 0, I_1 \land I_2 \vdash \Lambda_\delta(\Delta), I'_1 \land I'_2 \quad \text{(B.42)}$$

since $s^A - t^A \leq 0$ is equivalent to $s^A - t^A = 0$. Using Equations (B.42), (B.41) and the fact that two valid implications $\mu \rightarrow \nu$ and $\mu' \rightarrow \nu'$ entail the validity of $\mu \lor \mu' \rightarrow \nu \lor \nu'$ we deduce the validity of

$$\Lambda_\gamma(\Gamma), s^A - t^A \leq 0, (I_1 \land I_2) \lor I_3 \vdash \Lambda_\delta(\Delta), (I'_1 \land I'_2) \lor I'_3 \quad \text{(B.43)}$$

since $s^A - t^A \leq 0 \lor s^A - t^A \leq 0$ is equivalent to $s^A - t^A \leq 0$. For the case where $s^A = t^A$ we proceed similarly, i.e, in this case the functions in (B.35)-(B.38) all return the empty set $\emptyset$ thus implying the validity of

$$\Lambda_\gamma(\Gamma), I_1 \vdash \Lambda_\delta(\Delta), I'_1 \quad \text{(B.44)}$$

$$\Lambda_\gamma(\Gamma), I_2 \vdash \Lambda_\delta(\Delta), I'_2 \quad \text{(B.45)}$$

$$\Lambda_\gamma(\Gamma), I_3 \vdash \Lambda_\delta(\Delta), I'_3 \quad \text{(B.46)}$$

Using (B.44) and (B.45) we obtain

$$\Lambda_\gamma(\Gamma), (I_1 \land I_2) \lor I_3 \vdash \Lambda_\delta(\Delta), (I'_1 \land I'_2) \lor I'_3$$

Finally, using (B.46), (B.47) and (B.38) we again obtain the validity of (B.43).

For the rules FM-ELIM, SPLIT-EQ and SPLIT-NEQ, the proof follows the same lines.
This shows that any interpolating sequent in \( P \) and its relabeled version in \( P' \) are chain interpolating.

In particular, for a relabeled conclusion \( \Gamma, [\Gamma^*]_R \vdash [\Delta^*]_R, \Delta \upharpoonright I \) of any interpolating sequent \( \Gamma, [\Gamma^*]_L \vdash [\Delta^*]_L, \Delta \upharpoonright J \) with

\[
\gamma(\rho) := \rho \text{ for any } \rho \in \Gamma \text{ and} \\
\gamma([\phi]_R) := [\phi]_L \text{ for any } [\phi]_R \in [\Gamma^*]_R \\
\delta(\rho) := \rho \text{ for any } \rho \in \Delta \text{ and} \\
\delta([\phi]_R) := [\phi]_L \text{ for any } [\phi]_R \in [\Delta^*]_R
\]

we conclude that \( \Gamma^*, I \vdash \Delta^*, J \) is valid since \( \Lambda_\gamma(\Gamma \cup [\Gamma^*]_R) = \Gamma^* \) and \( \Lambda_\delta(\Delta \cup [\Delta^*]_R) = \Delta^* \).

\( \square \)
B.2 PA+UP is not Closed under Interpolation

Theorem 4.2.3. PA+UP is not closed under interpolation.

We give a proof of the theorem making use of the following intermediate result:

Lemma B.2.1. Let $y$ be a constant and $S = \{ \alpha_i y + \beta_i \mid \alpha_i, \beta_i \in \mathbb{Z}, i \in \{1, \ldots, n\} \}$ be a finite set of terms in PA. Then there exists an even number $a \in 2\mathbb{Z}$ such that 
\[
\frac{a}{2} \notin \{ \text{val}_{y\to a}(t) \mid t \in S \}.
\]

Proof. Choose $a \in 2\mathbb{Z}$ such that $a > 2 \cdot \max_{i} |\beta_i|$. Let us suppose that, for some $t = \alpha y + \beta \in S$, we have $\text{val}_{y\to a}(\alpha y + \beta) = \alpha a + \beta = \frac{a}{2}$. Thus $2\alpha a + 2\beta = a$ and therefore $(2\alpha - 1)a = -2\beta$. Since $2\alpha - 1 \neq 0$, we distinguish two cases:

- $2\alpha - 1 > 0$: this yields a contradiction because $(2\alpha - 1)a \geq a > 2 \cdot |\beta| = -2\beta$.
- $2\alpha - 1 < 0$: this yields a contradiction because $(2\alpha - 1)a \leq -a < -2 \cdot |\beta| = -2\beta$.

We can now prove Theorem 4.2.3.

Proof of Theorem 4.2.3. We construct an example of inconsistent formulae $A$ and $B$ in PA+UP whose interpolant requires quantification. Consider:

$$
A = 2c - y \not\equiv 0 \land p(c) \quad B = 2d - y \not\equiv 0 \land \neg p(d)
$$

The symbols $p$ and $y$ are common, while $c$ and $d$ are local. The conjunction $A \land B$ is unsatisfiable. The strongest and the weakest interpolants for $A$ and $B$ are, respectively:

$$
I_s = \exists x. (2x - y \not\equiv 0 \land p(x)) \quad I_w = \forall x. (2x - y \not\equiv 0 \rightarrow p(x))
$$

Now suppose $I$ is a quantifier-free interpolant for $A \land B$; in particular, $I$ contains only the common symbols $p$ and $y$. Let $S = \{ t \mid p(t) \text{ occurs in } I \}$ be the set of all terms occurring in $I$ as arguments of $p$. All elements of $S$ are PA terms
over the symbol y. By Lemma B.2.1, there is an even number $a \in 2\mathbb{Z}$ such that $\frac{a}{2} \not\in \{\text{val}_{y\to a}(t) \mid t \in S\}$.

Since $I$ is an interpolant, the implications $I_s \Rightarrow I$ and $I \Rightarrow I_w$ hold. In particular, observe that

$$\left(2 \mid y\right) \models (I_s \leftrightarrow I) \land (I \leftrightarrow I_w). \quad \text{(B.48)}$$

Choose an interpretation $K$ with $K(y) = a$ that satisfies $I$ (this is possible, because such satisfying interpretations exist for $I_s$). Because of (B.48) and because $K(y)$ is even, it holds that $\frac{K(y)}{2} \in K(p)$. However, we know that $I$ does not contain any atom $p(t)$ such that $\text{val}_K(t) = \frac{K(y)}{2}$. This means that $I$ is also satisfied by the interpretation $K'$ that coincides with $K$, with the only exception that $\frac{K'(y)}{2} \not\in K'(p)$. But $K'$ violates $I_w$, contradicting the assumption that $I$ is an interpolant. \qed

## B.3 Proofs generating PAID+UP Interpolants

We first give a proof of Lemma 4.2.7, from which Lemma 4.2.5 can be derived by providing a PAID+UP proof procedure. Theorem 4.2.6 then follows as a corollary.

### B.3.1 Sufficient Conditions for PAID+UP Interpolants

The only rules that introduce quantifiers in interpolants in $\mathcal{P}$ are 1. the rules close-eq-*, close-ineq, and 2. the rules all-left-* that are used to instantiate axiom (4.1). The quantifiers generated by the first kind of rules can directly be eliminated, because the body of the quantified expression is an arithmetic literal. In the second case, we consider the sub-proof $\mathcal{Q}$, as described in the lemma. There are different scenarios depending on the values of $D, E, F$; for sake of presentation, we only consider $D = L, E = R, F = L$ (all other cases are similar):
We need to analyse the shape of the interpolant

\[ J_5 = \forall R \bar{t} \ J_4 = \forall R \bar{t} \ (J_2 \lor p(\bar{t})) \]

(B.49)

where \( x_1, \ldots, x_n \) are all constants in \( \bar{t} \) that are \( R \)-local in the sequent

\[ \Gamma', [p(\bar{s})]_L \vdash [\bar{s} - \bar{t} \equiv 0]_L, [p(\bar{t})]_R, \Delta' \]

Using vector notation for \( \bar{x} = (x_1, \ldots, x_n)' \), the atom \( p(\bar{t}) \) can be represented as

\[ p(\bar{c} \bar{x} + \bar{v}) \]

where \( \bar{c}_1, \ldots, \bar{c}_n \in \mathbb{Z}^n \) are row vectors of coefficients, and \( \bar{v}_1, \ldots, \bar{v}_k \) are terms that do not contain any of the constants \( x_1, \ldots, x_n \). In matrix notation, this gives

\[ p(\bar{t}) = p(C \bar{x} + \bar{v}) \]

for \( C = (\bar{c}_1 \cdots \bar{c}_n)' \in \mathbb{Z}^{k \times n} \).

Because \( x_1, \ldots, x_n \) are \( R \)-local, we know that the constants do not occur in any partial interpolant in \( \Gamma', [p(\bar{s})]_L \vdash [\bar{s} - \bar{t} \equiv 0]_L, [p(\bar{t})]_R, \Delta' \). This implies that the term \(-\bar{c}_i \bar{x} \) in the partial interpolant of \( s_i - t_i \equiv 0 [s_i - t_i \equiv 0] \) will not be affected by any application of RED-RIGHT; likewise, applications of MUL-RIGHT can only introduce scaling by some factor \( \alpha \). It is therefore possible to represent the final partial interpolant \( u_i \equiv 0 \) in the form \( \alpha \bar{c}_i \bar{x} + u_i' \equiv 0 \), where \( \alpha \in \mathbb{Z} \setminus \{0\} \) and \( u_i' \) does not contain any of the constants \( x_1, \ldots, x_n \). This means that

\[ \bigvee_i u_i \not\equiv 0 \equiv -\bigwedge_i \alpha \bar{c}_i \bar{x} + u_i' \equiv 0 \equiv \alpha C \bar{x} + \bar{u}' \not\equiv 0 \]
We now consider the Smith decomposition [47] of the matrix $C$, i.e., the decomposition $C = LSR$ into three integer matrices, such that 1. $L \in \mathbb{Z}^{k \times k}$ and $R \in \mathbb{Z}^{n \times n}$ are invertible (over integers), 2. $S$ has the shape

$$
\begin{pmatrix}
\beta_1 & 0 & \ldots & \ldots & 0 \\
0 & \beta_2 & \ddots & & \\
& \ddots & \ddots & \ddots & \\
& & \ddots & \beta_r & 0 \\
0 & & & 0 & \ldots & 0
\end{pmatrix}
$$

where $r \leq \min\{k,n\}$ and $\beta_1, \ldots, \beta_r$ are positive integers such that $\beta_{i+1} \in \beta_i \mathbb{Z}$ for all $i \in \{1, \ldots, r-1\}$.

The interpolant $J_5$ in (B.49) can then be rewritten as follows:

$$J_5 \equiv \forall \bar{x}. \ (\alpha C \bar{x} + \bar{u}' \neq 0 \lor p(C \bar{x} + \bar{v}))$$

$$\equiv \forall \bar{x}. \ (\alpha LSR \bar{x} + \bar{u}' \neq 0 \lor p(LSR \bar{x} + \bar{v}))$$

$$\equiv \forall \bar{y}. \ (\alpha LS \bar{y} + \bar{u}' \neq 0 \lor p(LS \bar{y} + \bar{v}))$$

$$\equiv \forall y_1. \ (\alpha \beta_1 y_1 + (L^{-1} \bar{u}')_1 \neq 0 \lor$$

$$\forall y_2. \ (\alpha \beta_2 y_2 + (L^{-1} \bar{u}')_2 \neq 0 \lor$$

$$\vdots$$

$$\forall y_r. \ (\alpha \beta_r y_r + (L^{-1} \bar{u}')_r \neq 0 \lor p(LS \bar{y} + \bar{v})) \ldots)$$

$$\lor \bigvee_{i=r+1}^{k} (L^{-1} \bar{u}')_i \neq 0$$

where $\bar{y} = (y_1, \ldots, y_n)^t$ is a vector of fresh variables, and $(L^{-1} \bar{u}')_i$ denotes the $i$th element of the vector $L^{-1} \bar{u}'$ of terms. Note that the variables $y_{r+1}, \ldots, y_n$ only occur with coefficient zero in the expression $S \bar{y}$, and therefore do not have to be quantified. This shows that $J_5$ is equivalent to a PAID+UP formula and concludes the proof.
B.3. Completeness of the PAID+UP Calculus

We describe a proof procedure that, given an interpolating sequent \( \Gamma \vdash \Delta \downarrow \) such that \( \Gamma_L, \Gamma_R \vdash \Delta_L, \Delta_R \) is valid, generates a proof satisfying the conditions in Lemma 4.2.7. The following reasoning steps are performed:

1. Apply rules \( OR-* \), \( AND-* \), \( NOT-* \), \( EX-LEFT \), \( ALL-RIGHT \), \( ALL-LEFT-GRD \), \( EX-RIGHT-GRD \), \( DIV-* \), \( IPI-* \), \( SPLIT-* \) exhaustively; move all inequalities to the antecedent. This will eliminate all propositional connectives and quantifiers in formulae, what remains are proof goals of the form

\[
\left\{ t_i \leq 0 \left| t_i^A \leq 0 \right. \right\}_{i \in I}, \left\{ s_j \neq 0 \left| s_j^A \neq 0 \right. \right\}_{j \in J}, \vdash \{ \left| p_m(\bar{u}_m) \right|_{D_m} \}_{m \in M} \downarrow \?
\]

where \( I, J, K, M \) are disjoint sets of indexes.

2. Apply rules \( RED-LEFT \), \( COL-RED-* \), \( MUL-LEFT \) to solve the equalities in the antecedents, as described in [73]. This either leads to an unsatisfiable equality, in which case the rule \( CLOSE-EQ-LEFT \) can be applied, or to goals of the form

\[
\{ t_i \leq 0 \left| t_i^A \leq 0 \right. \}_{i \in I}, \left\{ \alpha_j c_j + s_j \neq 0 \left| s_j^A \neq 0 \right. \right\}_{j \in J}, \vdash \{ \left| p_m(\bar{u}_m) \right|_{D_m} \}_{m \in M} \downarrow \? \quad (B.50)
\]

where \( I, J, K, M \) are disjoint sets of indexes, \( \alpha_j \in \mathbb{Z} \setminus \{0\} \) divides all coefficients and constants terms in \( s_j \), the constants \( c_j \) are pairwise distinct, and no \( c_j \) occurs in any term \( t_i \) or \( s_j \). In particular, this means that the equalities \( \{ \alpha_j c_j + s_j \neq 0 \}_{j \in J} \) are satisfiable.

3. Whenever a sequent contains literals \( p_k(\bar{u}_k) \) and \( p_m(\bar{u}_m) \) such that \( p_k = p_m \), and such that \( \bar{u}_k \approx \bar{u}_m \) is implied by the equalities \( \{ \alpha_j c_j + s_j \neq 0 \}_{j \in J} \), instantiate the consistency axiom (4.1) for \( p_k(\bar{u}_k) \) and \( p_m(\bar{u}_m) \), and close the resulting sub-proofs as shown in Lemma 4.2.7 (i) and in the beginning of Section B.3.1.

4. Apply \( STRENGTHEN \) in a fair manner to the inequalities in the antecedents. Whenever a new equation is generated by a \( STRENGTHEN \) application, go
back to step 2. Whenever a sequent has been derived in which the inequalities in the antecedent are rationally inconsistent, apply FM-ELIM exhaustively, and apply CLOSE-INEQ to a resulting contradictory inequality.

This procedure will in finitely many steps construct a closed proof tree for the valid PAID+UP sequent $\Gamma \vdash \Delta$; by construction, the proof satisfies the conditions in Lemma 4.2.7 (i).

Two steps in the procedure require further considerations:

- **Termination of the loop 2–4:** it has to be shown that systematic application of STRENGTHEN terminates: on every branch of the generated proof, eventually a sequent is reached in which no inequalities remain, or in which the remaining inequalities are rationally inconsistent. Recall that every application of STRENGTHEN produces three new goals: one in which an inequality $t_i \leq 0$ has been turned into an equality $t_i \doteq 0$ (case (a)), and two in which an inequality $t_i \leq 0$ has been strengthened to $t_i + 1 \leq 0$ (case (b)).

Reasoning by contradiction, assume that the procedure never terminates on some branch. This means that, from some point on, we are always looking at the (b) case on the branch, and that the number of inequalities on the branch remains constant and non-zero.

Note that we can assume that each sequent (B.50) considered in step 4 is valid (ignoring interpolant annotations, which are not relevant at this point); equivalently, the following formula is unsatisfiable:

$$\bigwedge_{i \in I} t_i \leq 0 \land \bigwedge_{j \in J} \alpha_j c_j + s_j \doteq 0 \land \bigwedge_{k \in K, m \in M} \bar{u}_k \neq \bar{u}_m$$

By rewriting the negated equalities using the equalities $\alpha_j c_j + s_j \doteq 0$, eliminating every occurrence of a constant $c_j$, we obtain a new unsatisfiable conjunction without positive equalities:

$$\bigwedge_{i \in I} t_i \leq 0 \land \bigwedge_{k \in K, m \in M} \bar{u}_k' \neq \bar{u}_m'$$

Because step 3 has not been able to close the goal at hand, we can assume that each disjunction $\bar{u}_k' - \bar{u}_m' \neq 0$ contains at least one equality that is not
of the form $0 \neq 0$; we denote this equality with $v_{k,m} \neq 0$. We then know that also the following formula is unsatisfiable:

$$\bigwedge_{i \in I} t_i \leq 0 \land \bigwedge_{k \in K, m \in M} v_{k,m} \neq 0$$

This formula corresponds to the negation of formula (B.53) in Lemma B.4.1, which tells us that there is an $r \in \mathbb{R}$, and therefore also a $\beta \in \mathbb{Z}$, such that the following formula is even rationally unsatisfiable:

$$\bigwedge_{i \in I} t_i + \beta \leq 0$$

Because fair application of STRENGTHEN will eventually turn every inequality $t_i \leq 0$ into an inequality $t_i + \beta \leq 0$ such that $\beta_i \geq \beta$, it is guaranteed that the inequalities in the antecedent eventually become rationally unsatisfiable. This contradicts the assumption that the procedure does not terminate on the considered branch.

- **Existence of a complementary pair in step 3:** we have to show that a complementary pair of literals can be selected in step 3 once all inequalities have been eliminated from a sequent (in step 4). By assumption, we know that the sequents considered in step 3 are valid (again ignoring interpolants). Inequalities-less sequents of the form produced in step 2 are valid iff the sequent

$$\{ \alpha_j c_j + s_j \doteq 0 \}_{j \in J} \vdash \{ \bar{u}_k \doteq \bar{u}_m \}_{k \in K, m \in M}$$

is valid.

We reason by contradiction: suppose that each conjunction $\bar{u}_k \doteq \bar{u}_m$ of equalities contains one equation that is not implied by $\{ c_j + s_j / \alpha_j \doteq 0 \}_{j \in J}$; w.l.o.g., we can assume that this is always the first equation $u^1_k \doteq u^1_m$. This means that the sequents

$$\{ \alpha_j c_j + s_j \doteq 0 \}_{j \in J} \vdash u^1_k \doteq u^1_m$$

cannot be proven using the rules RED-RIGHT and MUL-RIGHT to reduce equalities in the succedent, and the rule CLOSE*-RIGHT to detect valid equalities. Consequently, the rules are not sufficient to prove the sequent

$$\{ \alpha_j c_j + s_j \doteq 0 \}_{j \in J} \vdash \{ u^1_k \doteq u^1_m \}_{k \in K, m \in M}$$
either. By completeness results in [71], this implies that the sequent is invalid. But then also (B.51) is invalid, contradicting the assumption.

**B.4 Integer Projection Lemma**

Let \( \{v_1, \ldots, v_n\} \) be a fixed set of variables. For any term \( t \), we introduce the function \( \tilde{t} : \mathbb{R}^n \to \mathbb{R} \) defined by \( \tilde{t}(x_1, \ldots, x_n) = [v_1/x_1, \ldots, v_n/x_n]t \).

**Lemma B.4.1.** Let \( \{t^1, \ldots, t^m\} \) be a set of terms of the form \( t^j = c_0^j + \sum_{i=1}^n c_i^j v_i \), and \( \{s^1, \ldots, s^p\} \) be a set of non-null terms of the form \( s^k = d_0^k + \sum_{i=1}^n d_i^k v_i \), i.e. for each \( k \), there exists an \( i \) with \( d_i^k \neq 0 \). Suppose the following formula is valid:

\[
\forall r \in \mathbb{R} \exists y_1, \ldots, y_n \in \mathbb{R} \forall j \in \{1, \ldots, m\} : t^j(y_1, \ldots, y_n) \leq r. \tag{B.52}
\]

Then the following formula is valid:

\[
\exists z_1, \ldots, z_n \in \mathbb{Z} (\forall j \in \{1, \ldots, m\} : \tilde{t}^j(z_1, \ldots, z_n) \leq 0 \land \forall k \in \{1, \ldots, p\} : s^k(z_1, \ldots, z_n) \neq 0 ). \tag{B.53}
\]

Before we can prove this lemma, we need a few definitions and auxiliary properties. Define the function \( f : \mathbb{R} \to \mathbb{R} \) via

\[
f(x_1, \ldots, x_n) = \max_{j=1}^m \tilde{t}^j(x_1, \ldots, x_n).
\]

We also define \( ||C|| = \sum_{i=1}^n \sum_{j=1}^m |c_{ij}| \), the 1-norm of the coefficient matrix induced by the \( t^j \).

**Property B.4.2.** For real numbers \( a, b, \varepsilon \), \( \max\{a + \varepsilon, b\} \leq \max\{a, b\} + |\varepsilon| \).

Proof:

1. If \( \varepsilon \geq 0 \), then \( a + \varepsilon \leq \max\{a, b\} + \varepsilon \), and \( b \leq b + \varepsilon \leq \max\{a, b\} + \varepsilon \). Thus \( \max\{a + \varepsilon, b\} \leq \max\{a, b\} + \varepsilon \leq \max\{a, b\} + |\varepsilon| \).

2. If \( \varepsilon < 0 \), then let \( \delta = -\varepsilon \geq 0 \). Using (i) with \( \delta \) in place of \( \varepsilon \), we obtain:

\[
\max\{a + \varepsilon, b\} \leq \max\{a + \delta, b\} \leq \max\{a, b\} + \delta = \max\{a, b\} + |\varepsilon|.
\]
Property B.4.3. Given \( y_1, \ldots, y_n \in \mathbb{R} \), define \( z_i := \lfloor y_i \rfloor \in \mathbb{Z} \) for \( i \in \{1, \ldots, n\} \), where \( \lfloor \cdot \rfloor \) denotes the floor of a real number. Then \( f(z_1, \ldots, z_n) \leq f(y_1, \ldots, y_n) + ||C|| \).

Proof: \[
f(z_1, \ldots, z_n) = \max_{j=1}^{m} \mathcal{t}_j(z_1, \ldots, z_n)
\]
\[
= \max_{j=1}^{m} \left( c_0^j + \sum_{i=1}^{n} c_i^j (y_i + z_i - y_i) \right)
\]
\[
= \max_{j=1}^{m} \left( c_0^j + \sum_{i=1}^{n} c_i^j y_i + \sum_{i=1}^{n} c_i^j (z_i - y_i) \right)
\]
\[
\leq \max_{j=1}^{m} \left( c_0^j + \sum_{i=1}^{n} c_i^j y_i \right) + \sum_{j=1}^{m} \left| \sum_{i=1}^{n} c_i^j (z_i - y_i) \right|
\]
\[
\leq f(y_1, \ldots, y_n) + \sum_{j=1}^{m} \sum_{i=1}^{n} |c_i^j| \cdot |z_i - y_i|
\]
\[
\leq f(y_1, \ldots, y_n) + ||C||
\]
where \( (\ast) \) applies property B.4.2 (\( m \) times), and \( (\ast\ast) \) uses \( |z_i - y_i| = |\lfloor y_i \rfloor - y_i| \leq 1 \).

Proof of Lemma B.4.1: Let \( r = -(p^2 + 1) \cdot ||C|| \). By (B.52), there exist \( y_1, \ldots, y_n \in \mathbb{R} \) such that \( f(y_1, \ldots, y_n) \leq r \). For \( 1 \leq i \leq n \), define \( g_i = \lfloor y_i \rfloor \in \mathbb{Z} \). Our final solutions \( z_i \) will have the form \( z_i = g_i + h_i \). We obtain suitable \( h_i \in \mathbb{Z} \) by examining the condition that the functions \( s^k \) must not evaluate to 0. The condition involving the \( \mathcal{t}_j \) will then be satisfied due to our choice of \( r \) above.

To this end, we prove that there exist integers \( h_1, \ldots, h_n \in \mathbb{Z}_{\geq 0} \) such that:

1. for all \( k \in \{1, \ldots, p\} \), \( s^k(g_1 + h_1, \ldots, g_n + h_n) \neq 0 \), and
2. for all \( i \in \{1, \ldots, n\} \), \( h_i \leq p^2 \).

The proof is by induction on \( p \):

- For \( p = 0 \), the claim holds trivially with \( h_i = 0 \) for all \( i \).
- Suppose the claim holds for \( p - 1 \). That is, we have for all \( k \in \{1, \ldots, p-1\} \), \( s^k(g_1 + h_1, \ldots, g_n + h_n) \neq 0 \), and \( h_i \leq (p - 1)^2 \) for all \( i \).
If $\overline{s^p}(g_1 + h_1, \ldots, g_n + h_n) \neq 0$, we can choose the same numbers $h_1, \ldots, h_n$.

If $\overline{s^p}(g_1 + h_1, \ldots, g_n + h_n) = 0$, let $i_0$ be such that $i_0 \neq 0$ and $d_{i_0}^p \neq 0$, i.e. $d_{i_0}^p$ is a non-zero coefficient of a variable in $\overline{s^p}$. Such an index exists since $s^p$ is non-null and $\overline{s^p}(g_1 + h_1, \ldots, g_n + h_n) = 0$ implies that $s^p$ cannot be the constant term $d_0^p$. We can now replace the argument $g_{i_0} + h_{i_0}$ by $g_{i_0} + h_{i_0} + 1$, in which case $\overline{s^p}$ will evaluate to $d_{i_0}^p$, which is non-zero, as desired. The problem is that this replacement may nullify a function $\overline{s^k_0}$ with $k_0 < p$. Note that this is only possible if $d_{i_0}^{k_0} \neq 0$, i.e. $\overline{s^k_0}$ must have a non-zero coefficient at the same position $i_0$ as $\overline{s^p}$. To re-enforce that $\overline{s^k_0}$ evaluates to non-zero, we replace $g_{i_0} + h_{i_0} + 1$ by $g_{i_0} + h_{i_0} + 2$. This replacement does not nullify $\overline{s^p}$ again, as the following argument shows: For $k \in \{1, \ldots, p\}$, $i \in \{1, \ldots, n\}$, integers $a_1, \ldots, a_n$ and $h \in \mathbb{Z}$:

$$\overline{s^k}(a_1, \ldots, a_{i-1}, a_i + h, a_{i+1}, \ldots, a_n) = \overline{s^k}(a_1, \ldots, a_n) + d_i^k \cdot h.$$ 

In particular, if $\overline{s^k}(a_1, \ldots, a_n) = 0$ and $d_i^k \neq 0$, then replacing the argument $a_i$ by any larger integer results in a non-zero value of $\overline{s^k}$. Thus, to complete the inductive step, we increase $g_{i_0} + h_{i_0}$ until none of the functions $\overline{s^k}$ evaluates to 0. Since there are $p$ such functions, this requires at most $p$ increases (of magnitude 1), thus $h_{i_0} \lessapprox (p-1)^2 + p \leq p^2$. Note that values $h_i$ with $i \neq i_0$ are not affected; here we simply have $h_i \leq (p-1)^2 \leq p^2$.

This concludes the inductive proof. What remains to show is that, with $z_i :=$
g_i + h_i for all i, we have \( f(z_1, \ldots, z_n) \leq 0 \):

\[
f(z_1, \ldots, z_n) = \max_{j=1}^{m} \left( c_0^j + \sum_{i=1}^{n} c_i^j (g_i + h_i) \right) \\
\leq \max_{j=1}^{m} \left( c_0^j + \sum_{i=1}^{n} c_i^j g_i \right) + \sum_{j=1}^{m} \left| \sum_{i=1}^{n} c_i^j h_i \right| \\
\leq f(g_1, \ldots, g_n) + \sum_{j=1}^{m} \sum_{i=1}^{n} |c_i^j| p^2 \\
= f(g_1, \ldots, g_n) + p^2 \cdot ||C|| \\
\leq (f(y_1, \ldots, y_n) + ||C||) + p^2 \cdot ||C|| \\
\leq (r + ||C||) + p^2 \cdot ||C|| \\
= 0,
\]

where (\(^*)\) applies property B.4.2 (\(m\) times), (\(\ast\ast\)) applies property B.4.3, and (\(\ast\ast\ast\)) is by the choice of \(r\).

## B.5 Proofs generating PAID+UF Interpolants

### B.5.1 Sufficient Conditions for PAID+UF \(_p\) Interpolants

The reasoning is similar as for Lemma 4.2.7 in Section B.3.1. Consider a sub-proof of \(\mathcal{P}\) of the form shown in Lemma 4.3.4. We only consider the case \(D = L, E = R, F = R\), as the other cases are similar. The interpolant generated by the sub-proof is

\[
J_6 = \exists_{L\check{s}_1 t_1} J_5 = \exists_{L\check{s}_1 t_1} J_5 = \exists_{L\check{s}_1 t_1} (J_3 \land f_p(\check{s}_1, t_1) \land J_4)
\]

By definition of PAID+UF \(_p\), we know that \(t_1\) is a Skolem constant. If \(t_1\) is \(L\)-local at this point in the proof (the quantifier \(\exists_{Lt_1}\) does not disappear), then it is \(L\)-local.
also in $\mathcal{Q}$, which means that $t_1$ does not occur in the interpolant $J_3$. This implies:

$$J_6 = \cdots \equiv \exists_{L\bar{s}_1} (J_3 \land \exists_{L t_1} (f_p(\bar{s}_1, t_1) \land J_4))$$

Furthermore, observe that the constants quantified by $\exists_{L\bar{s}_1}$ are $L$-local in $\mathcal{R}$, so that none of them occurs in $J_4$. As in the proof of Lemma 4.2.7, the expression $\exists_{L\bar{s}_1} (J_3 \land \cdots)$ can then be transformed to a sequence of guarded quantifiers:

$$J_6 \equiv \exists_{y_1}. (\alpha\beta_1 y_1 + (L^{-1}\bar{u}')_1 \equiv 0 \land \\
\exists_{y_2}. (\alpha\beta_2 y_2 + (L^{-1}\bar{u}')_2 \equiv 0 \land \\
\vdots \\
\exists_{y_r}. (\alpha\beta_r y_r + (L^{-1}\bar{u}')_r \equiv 0 \land \exists_{L t_1} (f_p(LS\bar{y} + \bar{v}, t_1) \land J_4)) \cdots)$$

$$\land \bigwedge_{i=r+1}^{k} (L^{-1}\bar{u}')_i \equiv 0$$

To conclude the proof, we have to consider two cases:

- $t_1$ is $L$-local and the quantifier $\exists_{L t_1}$ does not disappear: then we are finished, because it has been shown that $J_6$ is equivalent to a PAID+UF$_p$ formula.

- $\exists_{L t_1}$ disappears: in this case, we can rewrite the formula $f_p(LS\bar{y} + \bar{v}, t_1) \land J_4$ to $\exists_{z}. (f_p(LS\bar{y} + \bar{v}, z) \land t_1 \equiv z \land J_4)$, which is in PAID+UF$_p$.

### B.5.2 Closure of PAID+UF under Interpolation

Most importantly, we can first observe that a completeness result similar to Lemma 4.2.5 also holds for PAID+UF$_p$ (given in the next lemma). Theorem 4.3.2 then follows as a simple implication, because PAID+UF formulae can be translated to PAID+UF$_p$, interpolated, and the interpolant translated back to PAID+UF.

**Lemma B.5.1** (Completeness). Suppose that $A_{RE}, B_{RE}, \{ FC_f \}_{f \in F_A \cup F_B} \vdash$ is valid. Then there is a formula $I$ such that 1. the sequent

$$[A_{RE}]_L, [B_{RE}]_R, \{ [FC_f]_L \}_{f \in F_A}, \{ [FC_f]_R \}_{f \in F_B} \vdash \emptyset \triangleright I \quad (B.54)$$
is provable in the calculus of Section 4.2.1, enriched with the rules ALL-LEFT-GRD and EX-RIGHT-GRD, and 2. \( I \) is a PAID+UF\(_p\) formula up to normalisation of guards in expressions (4.2).

**Proof.** Given a sequent such that \( A_{RE}, B_{RE}, \{ FC_f \} \cup F_a \vdash \) is valid, we can construct a proof of (B.54) satisfying the conditions in Lemma 4.3.4 using a procedure similar to the one in Section B.3.2. Lemma 4.3.4 then guarantees that the interpolant \( I \) is in PAID+UF\(_p\) up to guard normalisation.

In the procedure of Section B.3.2, only step 3 has to be changed to obtain an algorithm for PAID+UF\(_p\): instead of searching for complementary literals \( p_k(\bar{u}_k) \) and \( p_m(\bar{u}_m) \), in the PAID+UF\(_p\) case we have to check for literals \( f_p(\bar{s}_1,t_1) \) and \( f_p(\bar{s}_2,t_2) \) such that \( \bar{s}_1 = \bar{s}_2 \) is implied by the equalities in the antecedent. If such a pair has been detected, the consistency axiom \( FC_f \) can be instantiated, closing the first three premises as dictated by Lemma 4.3.4. For the fourth premise, the proof procedure can go back to step 2 and continue proving. To ensure termination of the overall procedure, it only has to be guaranteed that the axiom \( FC_f \) is not repeatedly instantiated on the same branch for the same pair of literals \( f_p(\bar{s}_1,t_1) \) and \( f_p(\bar{s}_2,t_2) \). \( \square \)
Table C.1 shows additional details for the benchmarks presented in Figure 3.5. In this table, \#C is the number of constants and \#LC the number of local constants of the formula to be interpolated. IS denotes the size of the generated interpolants.

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>#C</th>
<th>#LC</th>
<th>Time(s)</th>
<th>IS</th>
<th>Time (s)</th>
<th>IS</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FISCHER10-2-fair.pri</td>
<td>33</td>
<td>11</td>
<td>42.831</td>
<td>11036</td>
<td>12.755</td>
<td>T/O</td>
<td></td>
</tr>
<tr>
<td>FISCHER11-2-fair.pri</td>
<td>36</td>
<td>12</td>
<td>81.914</td>
<td>21644</td>
<td>30.188</td>
<td>T/O</td>
<td></td>
</tr>
<tr>
<td>FISCHER1-2-fair.pri</td>
<td>6</td>
<td>2</td>
<td>.585</td>
<td>7</td>
<td>.164</td>
<td>58</td>
<td>.678</td>
</tr>
<tr>
<td>FISCHER1-3-fair.pri</td>
<td>8</td>
<td>2</td>
<td>.806</td>
<td>44</td>
<td>.152</td>
<td>14004</td>
<td>10.235</td>
</tr>
<tr>
<td>FISCHER1-4-fair.pri</td>
<td>10</td>
<td>3</td>
<td>1.297</td>
<td>69</td>
<td>.120</td>
<td>255945</td>
<td>253.898</td>
</tr>
<tr>
<td>FISCHER1-5-fair.pri</td>
<td>12</td>
<td>3</td>
<td>1.608</td>
<td>60</td>
<td>.142</td>
<td>302574</td>
<td>246.843</td>
</tr>
<tr>
<td>FISCHER1-6-fair.pri</td>
<td>14</td>
<td>5</td>
<td>2.102</td>
<td>78</td>
<td>.177</td>
<td>T/O</td>
<td></td>
</tr>
<tr>
<td>FISCHER2-2-fair.pri</td>
<td>9</td>
<td>3</td>
<td>.906</td>
<td>42</td>
<td>.141</td>
<td>1736</td>
<td>2.574</td>
</tr>
<tr>
<td>FISCHER2-3-fair.pri</td>
<td>12</td>
<td>2</td>
<td>3.124</td>
<td>311</td>
<td>.493</td>
<td>T/O</td>
<td></td>
</tr>
<tr>
<td>FISCHER2-4-fair.pri</td>
<td>15</td>
<td>2</td>
<td>7.872</td>
<td>730</td>
<td>1.350</td>
<td>T/O</td>
<td></td>
</tr>
<tr>
<td>FISCHER2-5-fair.pri</td>
<td>18</td>
<td>3</td>
<td>14.671</td>
<td>1578</td>
<td>2.075</td>
<td>T/O</td>
<td></td>
</tr>
<tr>
<td>FISCHER2-6-fair.pri</td>
<td>21</td>
<td>4</td>
<td>34.243</td>
<td>3271</td>
<td>5.568</td>
<td>T/O</td>
<td></td>
</tr>
<tr>
<td>FISCHER3-2-fair.pri</td>
<td>12</td>
<td>4</td>
<td>1.549</td>
<td>128</td>
<td>.188</td>
<td>2705</td>
<td>3.344</td>
</tr>
<tr>
<td>FISCHER3-3-fair.pri</td>
<td>16</td>
<td>3</td>
<td>10.909</td>
<td>1420</td>
<td>1.949</td>
<td>T/O</td>
<td></td>
</tr>
<tr>
<td>FISCHER3-4-fair.pri</td>
<td>20</td>
<td>4</td>
<td>26.780</td>
<td>2352</td>
<td>4.764</td>
<td>T/O</td>
<td></td>
</tr>
<tr>
<td>FISCHER3-5-fair.pri</td>
<td>24</td>
<td>6</td>
<td>79.662</td>
<td>8495</td>
<td>21.130</td>
<td>T/O</td>
<td></td>
</tr>
<tr>
<td>FISCHER4-2-fair.pri</td>
<td>15</td>
<td>5</td>
<td>2.174</td>
<td>248</td>
<td>.365</td>
<td>5406</td>
<td>5.292</td>
</tr>
<tr>
<td>FISCHER4-3-fair.pri</td>
<td>20</td>
<td>4</td>
<td>15.742</td>
<td>1869</td>
<td>3.183</td>
<td>T/O</td>
<td></td>
</tr>
<tr>
<td>FISCHER4-4-fair.pri</td>
<td>25</td>
<td>6</td>
<td>150.271</td>
<td>19194</td>
<td>45.608</td>
<td>T/O</td>
<td></td>
</tr>
<tr>
<td>FISCHER5-2-fair.pri</td>
<td>18</td>
<td>6</td>
<td>3.608</td>
<td>423</td>
<td>.719</td>
<td>22925</td>
<td>32.137</td>
</tr>
<tr>
<td>FISCHER5-3-fair.pri</td>
<td>24</td>
<td>6</td>
<td>37.412</td>
<td>4983</td>
<td>12.070</td>
<td>T/O</td>
<td></td>
</tr>
<tr>
<td>FISCHER5-4-fair.pri</td>
<td>30</td>
<td>8</td>
<td>190.332</td>
<td>20861</td>
<td>75.351</td>
<td>T/O</td>
<td></td>
</tr>
</tbody>
</table>
## Appendix C. Detailed Experimental Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#C</th>
<th>#LC</th>
<th>Time(s)</th>
<th>PRINCESS</th>
<th>QE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Time (s)</td>
<td>IS</td>
<td>IS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Time (s)</td>
<td>IS</td>
</tr>
<tr>
<td>Benchmark Name</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FISCHER6-2-fair.pri</td>
<td>21</td>
<td>7</td>
<td>5.453</td>
<td>1029</td>
<td>1.342</td>
</tr>
<tr>
<td>FISCHER6-3-fair.pri</td>
<td>28</td>
<td>12</td>
<td>66.319</td>
<td>6327</td>
<td>18.813</td>
</tr>
<tr>
<td>FISCHER7-2-fair.pri</td>
<td>24</td>
<td>8</td>
<td>6.870</td>
<td>1511</td>
<td>1.548</td>
</tr>
<tr>
<td>FISCHER7-3-fair.pri</td>
<td>32</td>
<td>16</td>
<td>211.202</td>
<td>14803</td>
<td>68.061</td>
</tr>
<tr>
<td>FISCHER8-2-fair.pri</td>
<td>27</td>
<td>9</td>
<td>14.738</td>
<td>2954</td>
<td>3.041</td>
</tr>
<tr>
<td>FISCHER8-3-fair.pri</td>
<td>36</td>
<td>18</td>
<td>304.832</td>
<td>25242</td>
<td>136.948</td>
</tr>
<tr>
<td>FISCHER9-2-fair.pri</td>
<td>30</td>
<td>10</td>
<td>25.428</td>
<td>6760</td>
<td>7.960</td>
</tr>
<tr>
<td>FISCHER9-3-fair.pri</td>
<td></td>
<td></td>
<td>T/O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULTIPL_2.msat.pri</td>
<td>14</td>
<td>8</td>
<td>1.104</td>
<td>23</td>
<td>.310</td>
</tr>
<tr>
<td>MULTIPL_3.msat.pri</td>
<td>18</td>
<td>11</td>
<td>2.001</td>
<td>50</td>
<td>1.127</td>
</tr>
<tr>
<td>MULTIPL_4.msat.pri</td>
<td>22</td>
<td>14</td>
<td>4.465</td>
<td>112</td>
<td>2.406</td>
</tr>
<tr>
<td>MULTIPL_5.msat.pri</td>
<td>30</td>
<td>20</td>
<td>30.660</td>
<td>O/M</td>
<td>27893</td>
</tr>
<tr>
<td>MULTIPL_6.msat.pri</td>
<td>34</td>
<td>23</td>
<td>121.682</td>
<td>133772</td>
<td>219.825</td>
</tr>
<tr>
<td>MULTIPL_7.msat.pri</td>
<td></td>
<td></td>
<td>T/O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULTIPL_8.msat.pri</td>
<td></td>
<td></td>
<td>336661</td>
<td>200.530</td>
<td></td>
</tr>
<tr>
<td>SIMPLEBIT_..2.pri</td>
<td>12</td>
<td>6</td>
<td>1.617</td>
<td>38</td>
<td>.369</td>
</tr>
<tr>
<td>SIMPLEBIT_..3.pri</td>
<td>15</td>
<td>9</td>
<td>3.611</td>
<td>187</td>
<td>.822</td>
</tr>
<tr>
<td>SIMPLEBIT_..4.pri</td>
<td>18</td>
<td>12</td>
<td>8.046</td>
<td>1001</td>
<td>1.808</td>
</tr>
<tr>
<td>SIMPLEBIT_..5.pri</td>
<td>21</td>
<td>15</td>
<td>22.661</td>
<td>3185</td>
<td>5.011</td>
</tr>
<tr>
<td>SIMPLEBIT_..6.pri</td>
<td>24</td>
<td>18</td>
<td>80.208</td>
<td>11294</td>
<td>18.079</td>
</tr>
<tr>
<td>prp-2-17.pri</td>
<td>233</td>
<td>180</td>
<td>4.537</td>
<td>92</td>
<td>.820</td>
</tr>
<tr>
<td>prp-2-19.pri</td>
<td>378</td>
<td>293</td>
<td>54.696</td>
<td>4451</td>
<td>16.404</td>
</tr>
<tr>
<td>prp-3-18.pri</td>
<td>278</td>
<td>216</td>
<td>10.202</td>
<td>591</td>
<td>2.853</td>
</tr>
<tr>
<td>prp-3-19.pri</td>
<td>378</td>
<td>294</td>
<td>36.130</td>
<td>634</td>
<td>8.537</td>
</tr>
<tr>
<td>prp-4-19.pri</td>
<td>378</td>
<td>293</td>
<td>13.645</td>
<td>288</td>
<td>3.311</td>
</tr>
<tr>
<td>exp10_3vars_0ite ... .pri</td>
<td>27</td>
<td>25</td>
<td>2.681</td>
<td>14</td>
<td>.608</td>
</tr>
<tr>
<td>exp10_3vars_1ite ... .pri</td>
<td>27</td>
<td>25</td>
<td>2.625</td>
<td>13</td>
<td>.450</td>
</tr>
<tr>
<td>exp10_3vars_2ite ... .pri</td>
<td>27</td>
<td>25</td>
<td>2.696</td>
<td>11</td>
<td>.352</td>
</tr>
<tr>
<td>exp10_4vars_0ite ... .pri</td>
<td>40</td>
<td>38</td>
<td>35.611</td>
<td>20</td>
<td>4.322</td>
</tr>
<tr>
<td>exp10_4vars_1ite ... .pri</td>
<td>40</td>
<td>38</td>
<td>38.907</td>
<td>19</td>
<td>3.649</td>
</tr>
<tr>
<td>exp10_4vars_2ite ... .pri</td>
<td>40</td>
<td>38</td>
<td>39.506</td>
<td>18</td>
<td>5.690</td>
</tr>
<tr>
<td>exp10_4vars_3ite ... .pri</td>
<td>40</td>
<td>38</td>
<td>29.398</td>
<td>17</td>
<td>2.760</td>
</tr>
<tr>
<td>exp12_3vars_0ite ... .pri</td>
<td>27</td>
<td>25</td>
<td>2.675</td>
<td>14</td>
<td>.549</td>
</tr>
<tr>
<td>exp12_3vars_1ite ... .pri</td>
<td>27</td>
<td>25</td>
<td>2.675</td>
<td>14</td>
<td>.549</td>
</tr>
<tr>
<td>exp12_3vars_2ite ... .pri</td>
<td>27</td>
<td>25</td>
<td>2.616</td>
<td>13</td>
<td>.425</td>
</tr>
<tr>
<td>exp12_3vars_3ite ... .pri</td>
<td>27</td>
<td>25</td>
<td>2.616</td>
<td>13</td>
<td>.425</td>
</tr>
<tr>
<td>exp12_4vars_0ite ... .pri</td>
<td>40</td>
<td>38</td>
<td>35.788</td>
<td>20</td>
<td>4.240</td>
</tr>
<tr>
<td>exp12_4vars_1ite ... .pri</td>
<td>40</td>
<td>38</td>
<td>37.994</td>
<td>19</td>
<td>3.445</td>
</tr>
<tr>
<td>exp12_4vars_2ite ... .pri</td>
<td>40</td>
<td>38</td>
<td>40.085</td>
<td>18</td>
<td>5.718</td>
</tr>
<tr>
<td>exp12_4vars_3ite ... .pri</td>
<td>40</td>
<td>38</td>
<td>29.515</td>
<td>17</td>
<td>2.774</td>
</tr>
<tr>
<td>exp14_3vars_0ite ... .pri</td>
<td>27</td>
<td>25</td>
<td>2.687</td>
<td>11</td>
<td>.349</td>
</tr>
<tr>
<td>exp14_3vars_1ite ... .pri</td>
<td>27</td>
<td>25</td>
<td>2.687</td>
<td>11</td>
<td>.349</td>
</tr>
<tr>
<td>exp14_3vars_2ite ... .pri</td>
<td>27</td>
<td>25</td>
<td>2.720</td>
<td>11</td>
<td>.359</td>
</tr>
<tr>
<td>exp14_4vars_0ite ... .pri</td>
<td>40</td>
<td>38</td>
<td>35.925</td>
<td>20</td>
<td>4.340</td>
</tr>
<tr>
<td>exp14_4vars_1ite ... .pri</td>
<td>40</td>
<td>38</td>
<td>37.799</td>
<td>19</td>
<td>3.484</td>
</tr>
<tr>
<td>exp14_4vars_2ite ... .pri</td>
<td>40</td>
<td>38</td>
<td>40.463</td>
<td>18</td>
<td>5.750</td>
</tr>
<tr>
<td>exp14_4vars_3ite ... .pri</td>
<td>40</td>
<td>38</td>
<td>29.256</td>
<td>17</td>
<td>2.857</td>
</tr>
<tr>
<td>Benchmark</td>
<td>Name</td>
<td>#C</td>
<td>#LC</td>
<td>Time(s)</td>
<td>PRINCESS IS</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------------</td>
<td>----</td>
<td>-----</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>exp16_3vars_0ite</td>
<td>27</td>
<td>25</td>
<td>2.656</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>exp16_3vars_1ite</td>
<td>27</td>
<td>25</td>
<td>2.644</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>exp16_3vars_2ite</td>
<td>27</td>
<td>25</td>
<td>2.737</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>exp16_4vars_0ite</td>
<td>40</td>
<td>38</td>
<td>36.331</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>exp16_4vars_1ite</td>
<td>40</td>
<td>38</td>
<td>38.062</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>exp16_4vars_2ite</td>
<td>40</td>
<td>38</td>
<td>39.812</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>exp16_4vars_3ite</td>
<td>40</td>
<td>38</td>
<td>29.588</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>exp4_3vars_0ite</td>
<td>27</td>
<td>25</td>
<td>2.622</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>exp4_3vars_1ite</td>
<td>27</td>
<td>25</td>
<td>2.631</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>exp4_3vars_2ite</td>
<td>27</td>
<td>25</td>
<td>2.731</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>exp4_4vars_0ite</td>
<td>40</td>
<td>38</td>
<td>27.009</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>exp4_4vars_1ite</td>
<td>40</td>
<td>38</td>
<td>22.897</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>exp4_4vars_2ite</td>
<td>40</td>
<td>38</td>
<td>23.336</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>exp4_4vars_3ite</td>
<td>40</td>
<td>38</td>
<td>25.475</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>exp4_5vars_0ite</td>
<td>53</td>
<td>51</td>
<td>128.546</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>exp4_5vars_1ite</td>
<td>53</td>
<td>51</td>
<td>128.848</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>exp4_5vars_2ite</td>
<td>53</td>
<td>51</td>
<td>134.831</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>exp4_5vars_3ite</td>
<td>53</td>
<td>51</td>
<td>98.794</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>exp4_5vars_4ite</td>
<td>55</td>
<td>53</td>
<td>123.085</td>
<td>22</td>
</tr>
</tbody>
</table>

Table C.1: Comparison of the interpolation procedure to QE
Bibliography


Curriculum Vitae

Personal

Angelo Brillout
Born December 8th 1980 in Paris, France

Education

2000 – 2002 Vordiplom from the University of Karlsruhe, Germany
2002 – 2005 Master of Science from the INSA de Lyon, France
2005 – 2006 Diplom-Informatiker from the University of Karlsruhe, Germany
2007 – 2011 PhD Student in the Formal Verification Group of Prof D. Kröning, ETH Zürich, Switzerland

Professional Experience

2003 Intern at DEKRA, Stuttgart, Germany
2004 Intern at EADS Astrium, Friedrichshafen, Germany
2005 Research Assistant at Forschungszentrum für Informatik, Karlsruhe, Germany
2007 – 2009 Teaching Assistant at ETH Zürich, Switzerland