JOHANNES KEPLER
UNIVERSITÄT LINZ
JKU

TNF
Technisch-Naturwissenschaftliche
Fakultät

Computational Logic and Quantifier Elimination Techniques for (Semi-)automatic Static Analysis and Synthesis of Algorithms

DISSERTATION

zur Erlangung des akademischen Grades

Doktorin

im Doktoratsstudium der

Technischen Wissenschaften

Eingereicht von:
Mădălina Erașcu

Angefertigt am:
Research Institute for Symbolic Computation

Beurteilung:
Prof. Dr. Tudor Jebelean (Betreuung)
Prof. Dr. Hoon Hong

Linz, November 2012

Research Institute for Symbolic Computation
Johannes Kepler University Linz, Austria

Computational Logic and Quantifier Elimination Techniques for (Semi-)automatic Static Analysis and Synthesis of Algorithms

Mădălina Eraşcu

Doctoral Thesis, November 2012

Research Institute for Symbolic Computation
Johannes Kepler University Linz, Austria

Computational Logic and Quantifier Elimination Techniques for (Semi-)automatic Static Analysis and Synthesis of Algorithms

Mădălina Eraşcu

Doctoral Thesis

advised by

Prof. Dr. Tudor Jebelean
Prof. Dr. Hoon Hong

Defended on December 12, 2012, Linz, Austria.

The work was supported by:

- Upper Austrian Government (October 2008 - July 2009, November 2009, December 2012)
- Austrian Ministry of Research (August - October 2009)
- Austrian Science Fund (FWF): W1214-N15, project DK1 (December 2009 - December 2010)
- Marshall Plan Foundation Scholarship (January 2011 – May 2011)
- DOC-fFORTE-fellowship of the Austrian Academy of Sciences (June 2011 – November 2012)

Abstract

This thesis presents logical and algebraic approaches for analyzing imperative recursive algorithms and for synthesizing optimal algorithms.

First we develop, formalize, and prove automatically, in the *Theorema* system (`www.theorema.org`), the soundness of a method for the verification of imperative recursive programs. Our goal is to identify the minimal logical apparatus necessary for formulating and proving (in computer-assisted manner) a correct collection of methods for program verification. Our work shows that reasoning about programs does not necessarily need a complex theoretical construction, because it is possible to transfer the semantics of the program into the semantics of logical formulas, thus avoiding any special theory related to program execution. We express the semantics as an implicit definition, in the object theory, of the function implemented by the program. Termination, defined also in the object theory, is an induction principle developed from the structure of the program with respect to iterative structures (recursive calls and `while` loops). An object theory is the theory relevant to the predicates, constants, and functions occurring in the program text. Currently, our method can be applied to programs with single recursion and with arbitrarily-nested loops with abrupt termination (`break`, `return`).

Second we investigate methods for synthesizing optimal algorithms. As a case study, we consider the square root problem: given the real number x and the error bound ε, find a real interval such that it contains $\sqrt{x}$ and its width is less than ε. We use iterative refining as algorithm schema: the algorithm starts with an initial interval and repeatedly updates it by applying a refinement map, say f, on it until it becomes narrow enough. Then the synthesis amounts to finding a refinement map f that ensures that the algorithm is correct (loop invariant), terminating (contraction), and optimal. All these could be formulated as quantifier elimination over the real numbers. Hence, in principle, they could be performed automatically. However, the computational requirement is huge, making the automatic synthesis practically impossible with the current general quantifier elimination software. Therefore, we performed some hand derivations and were able to synthesize semi-automatically optimal algorithms under natural assumptions.

Keywords: program analysis, imperative recursive programs, abrupt termination, Theorema system, program synthesis, square root computation, quantifier elimination

Zusammenfassung

In dieser Arbeit werden logische und algebraische Zugänge zur Analyse von imperativen-rekursiven Algorithmen sowie zur Synthese von optimalen Algorithmen präsentiert.

Zunächst entwickeln und formalisieren wir eine Methode zur Verifikation von imperativen-rekursiven Programmen, die automatisch mit dem System *Theorema* (`www.theorema.org`) bewiesen wird. Unser Ziel ist es das minimale logische Gerüst zu bestimmen, das notwendig ist um eine korrekte Sammlung von Methoden für Programmverifikation zu formulieren und (computerunterstützt) zu beweisen. Unsere Arbeit zeigt, dass für das Schlußfolgernüber Programme nicht notwendigerweise eine komplexe theoretische Konstruktion benötigt wird, da es möglich ist die Semantik des Programms in die Semantik logischer Formelnüberzuführen und damit spezielle Theorien über die Exekution von Programmenvermieden werden können. Die Semantik wird, in der Objekttheorie, als implizite Definition der Funktion, die durch das Programm implementiert ist, dargestellt. Termination, ebenfalls in der Objekttheorie definiert, ist ein Induktionsprinzip gebildetvon der Struktur des Programms bezüglich iterativer Strukturen (rekursive Aufrufe und `while`-Schleifen). Eine Objekttheorie ist die Theorie über die Prädikate, Konstanten und Funktionen, die im Programmtext vorkommen. Derzeit kann unsere Methode auf Programme mit einer einfachen Rekursion und mit beliebig verschachtelten Schleifen mit abruptem Abbruch (`break`, `return`) angewandt werden.

Im zweiten Teil untersuchen wir Methoden zur Synthese von optimalen Algorithmen. Als ein Fallbeispiel betrachten wir das Quadratwurzelproblem: gegeben eine reelle Zahl x undeine Fehlerschranke ε, ist eine reelles Intervall zu bestimmen, das x enthält und dessen Länge kleiner als ε ist. Als Schema für den Algorithmus verwenden wir iterative Verfeinerung: der Algorithmus startet mit einem Anfangsintervall, das wiederholt aktualisiert wird durch die Anwendung einer Verfeinerungsabbildung, nennen wir sie f, solange bis es klein genug ist. In diesem Fall entspricht die Synthese dem Bestimmen einer Verfeinerungsabbildung f, die sicher stellt, dass der Algorithmus korrekt ist (schleifeninvariant), terminiert (kontrahierend) und optimal ist. Diese Anforderungen können als Quantoreneliminationsproblem über den reellen Zahlen formuliert werden. Daher könnte das Problem laut Theorie automatischgelöst werden. In der Praxis sind die Rechenanforderungen zu immens und somit ist die automatische Synthese derzeit mit der aktuell verfügbaren Software für Quantorenelimination nicht durchführbar. Deshalb wurden einige Umformungsschritte von Hand ausgeführt und wir haben in einem semi-automatischen Prozess optimale Algorithmen (unter natürlichen Voraussetzungen) synthetisiert.

Stichworte: Programmanalyse, imperative rekursive Programme, abrupte Termination, Theorema, Programmsynthese, Quadratwurzelberechnung, Quantorenelimination

Acknowledgements

Foremost, I thank Professor Tudor Jebelean. Through his lectures and meetings, he introduced me, already during the Master's studies, to the fields of automated theorem proving and program verification. Afterwards, he guided me attentively during the PhD studies, and most important, encouraged me exactly at the right moment. Thank you very much for your clear explanations, permanent encouragements, infinite patience, help and support!

I thank Professor Hoon Hong for the interesting research problem he proposed and his guidance in finding the answer to it. He also hosted me at North Carolina State University (NCSU), Raleigh, USA, where I basically got infected with his contagious enthusiasm on doing research. Once I returned to Austria, he still continued to answer my questions and correct the errors of my writings. I thank him for his patience and also for all the effort he put in transforming me in a young professional researcher.

Professor Buchberger made possible, through his generous financial support, my participation at various conferences, workshops, and summer schools. This way I was able to make myself known to the community and to develop my network. I thank him for that. Also, many thanks, for the numerous comments and suggestions for improvement of my research work, writings, and presentations.

Wolfgang Schreiner gave many interesting lectures on various formal methods topics which I enjoyed a lot, except, sometimes, the difficult homeworks. He was also interested in the topic of my PhD research, asked a lot of questions and gave a lot of good ideas for improvement. He helped me with German translations and polishing the DOC-fFORTE proposal. Thank you!

I thank Professor Ali Nesin for giving me the possibility to teach in the Math Village, such a nice place to be, and for introducing me to the Turkish hospitality.

I thank Laura Kovacs for her suggestions and comments which helped a lot at the finalization and acceptance of the DOC-fFORTE proposal.

At the beginning of my PhD studies, Silviu Radu made me understand better computer algebra and algebraic combinatorics. Besides, Silviu always provided help of any kind, even before asking. Thank you, Pusicule!

I thank Veronika Pillwein for the German translation of the abstract of this thesis and for her struggle in explaining me German grammar and vocabulary. After all, I still speak German at the level of a child. But it is not my fault, German is too complex!

I thank the secretaries Tanja Gutenbrunner and Gabriela Hahn for depriving me from headaches that administration stuff could have provoked.

During my PhD studies, I have spent seven months in the US, at NCSU. My stay there would not have been so enjoyable without the help of Ms Pauline Hong (Thank you for providing tasty and healthy food!), Holly Durham (Thank you for the outings, rides to the grocery store, and the hot shower :)!) and the Regalados (Thank you for the peruvian food and hospitality!). I also thank the colleagues at NCSU, especially Clemens (a former RISC colleague who was also in exile there that time), Daniel, Ismail, Matt, Minsung, and Tulay, for the time spend together.

I thank my friends Ionela, Isabela, Nebiye-Bacım, and Faranak-Penguin for being so nice to me. I also thank Ionela for providing the LaTeX template of this thesis. I should not forget the three musketeers, Hamid, Max, and Zaf, for the RISC-like atmosphere in the Math Village and for the sightseeing of the "stones", Johannes, for being a devoted Sushi mate, and Anja, for the home-made cake.

I thank Father Sorin Bugner from the Romanian-Orthodox Church from Linz for the wonderful Sunday Liturgies and his spiritual guidance during all these years.

Last but not least, I thank my family (Iti, tati, buni, Iulian, Luiza) for their support from the distance, and Daniel for making me grow as a person in so many ways. Above all, I thank God for guiding me in this journey.

Contents

1. Introduction

Exactly ten years ago, a study conducted by Department of Commerce – National Institute of Standards and Technology (NIST) concluded that "software bugs, or errors, are so prevalent and so detrimental that they cost the U.S. economy an estimated 59.5 billion annually, or about 0.6 percent of the gross domestic product." [22]. Therefore, the desire is to write error-free programs. For the achievement of this goal, the following techniques are combined: *programming language design*, *debugging*, and *testing*. *Programming language design* represents the main step in writing correct software. The features of the nowadays programming languages (type systems, abstract data types, inheritance and encapsulation for object oriented programming, etc.) allow writing software at high level of abstraction and implicitly reducing the number of possible errors. By *debugging*, one can reduce the number of bugs in a software program in a systematic way such that it behaves as expected. *Testing* is an empirical step towards software verification. It is performed with the intention of finding software bugs but it can not provide the certainty of software correctness.

Nighter of these techniques, nor their combination, give a software correctness proof.

Program verification and *program synthesis* are able to prove, respectively, to generate correct software. *Program (formal) verification* is the technique which ensures or disproves the correctness of a computer program with respect to a specification. By *synthesis*, new programs are discovered which are known to be correct-by-construction.

We are interested in verifying and synthesizing programs at the implementation level, using *computational logic* and *quantifier elimination (QE) techniques*. More precisely, in verification, we want to solve the following problem: *given* a program augmented with specification (input and output condition), *generate* the verification conditions which ensure the fact that the program fulfills its specification. To solve this problem, we develop a method based on forward symbolic execution and functional semantics and we prove automatically the soundness of it, namely if the verification conditions hold, then the program is totaly correct with respect to its specification. In synthesis, we approach the problem of *discovering* the optimal algorithms of a *given* program scheme, which is terminating and fulfills certain annotations (specification and loop invariant).

On one hand, in this thesis, we focus on the automation of proving methods for ensuring the soundness of the verification conditions generator. The soundness and relative completeness proofs of the classical Hoare logic are well-established for sequential imperative programming languages. The same holds for logics extending Hoare logic with recursive calls, abrupt termination, exceptions, object-oriented features, etc. [45], [4], [68], [86]. These proofs are mainly done by defining the semantics in type theory [86] and by using the proof assistants Coq [7], Isabelle/HOL [67], PVS [70] in the (interactive) proofs. In a functional setting, the most common way of defining the semantics of the programs is to use Scott fixed-point theory [82]. Additionally to the definition of semantics, these proofs require to define the notion of termination. For imperative languages:

1. if the semantics defines a memory model of the program then inference rules for both partial correctness and termination are introduced,
2. if an axiomatic semantics is defined then inference rules for termination are defined only for iterative structures [1]; these inference rules capture the well-foundedness property of the iterative structure.

In the functional setting, termination is defined as being the least fix-point of certain recursive operators. Needless to say, the complexity of the proofs of the program logics depends on the choice of the semantics and the definition of termination. The complexity plays a crucial role especially if one aims at the automatization of such proofs. We try to avoid complex proofs by defining the semantics and the termination in the same logic as the one of the program.

Our work deals with automatic proof of soundness, in the *Theorema* system [27–29], of a method handling imperative recursive programs. The method is based on forward symbolic execution [51] and functional semantics [61]. Our main aim is the identification of the minimal logical apparatus necessary for formulating and proving (in a computer-assisted manner) a correct collection of methods for program verification. Our work shows that reasoning about imperative recursive programs does not necessarily need a complex theoretical construction, because it is possible to transfer the semantics of the program into the semantics of the logical formulas, thus avoiding any special theory related to program execution. Moreover, even the termination condition can be expressed as a logical formula in the object theory of the domain manipulated by the program. In our approach, this condition is in fact equivalent to an induction principle, which makes it very instrumental in proving the existence and uniqueness of the function implemented by the loop.

On the other hand, in this thesis, we focus on synthesizing optimal and reliable numeric algorithms. We consider a case study, namely computing the square root of a real number: given the real number x and the error bound ε, we are searching for a real interval such that it contains $\sqrt{x}$ and its width is less than ε. We fix the algorithm schema, namely, iterative refining: the algorithm starts with an initial interval and repeatedly updates it by applying a refinement function, say f, on it until it becomes narrow enough. The synthesis amounts to finding a refinement function f that ensures that the algorithm is correct (loop invariant), terminating (contraction),

[1] We call *iterative structure* a recursive call or a loop.

and optimal. All these can be formulated as QE problems over the real numbers. Hence, in principle, they can be all carried out automatically. However, the computational requirement is so huge, making the automatic synthesis practically impossible with the state-of-the-art QE software. Hence, we did some hand derivations and were able to synthesize semi-automatically optimal algorithms under suitable assumptions. Initially, we considered a well-known refinement function which solves the problem, namely Secant-Newton. It is known that the Secant-Newton refinement function has quadratic convergence, the same as the Newton algorithm, the benefit is that it does not require an initial estimate of the solution. We proved that the Secant-Newton refinement function is optimal among all its natural generalizations, that is, among functions which are contracting and are quadratic rational functions in the end points of the input interval. Further, we proved that all natural generalizations of Secant-Newton function, including the function itself, have the best Lipschitz constant $\frac{1}{2}$. Furthermore, by dropping off the contraction condition of the refinement function and imposing other natural assumptions on it (the Secant-Newton refining function satisfies these constraints), we were able to synthesize semi-automatically new refining functions for which the best Lipschitz constant is $\frac{1}{4}$.

1.1. Related Work

Research into program *analysis* [32, 39] and *synthesis* [24, 35] has a long tradition, however, in the last two decades a tremendous advance of techniques is noticed due to increasing usage of computers in human's life.

Our approach in *program analysis* follows the principles of forward symbolic execution [51] and functional semantics [61], but additionally gives formal definitions in a meta-theory for the meta-level functions which define the syntax, the semantics, and the verification conditions. To our knowledge there is no other work on symbolic execution approaching the verification problem in a fully formal way. However, the ideas from the formalization of the calculus are not completely new; [55] describes the behavior of concurrent systems as relation between the variables in the current state and in the post-state. A similar approach is encountered in [8] where the program equations (involving relation between current and post-state) are used to express nondeterminacy and termination. In the same manner, [76] presents the formal calculus for imperative languages containing complex structures. Specification languages used in the framework of verification tools also use this concept – see e.g. JML [56].

Program logics for reasoning about programs with abrupt statements are implemented in many state-of-the-art program verifiers. In the KeY system [4], which has a first-order predicate programming logic (called Java Card DL) with subtyping extended with parameterized modal operators, there are two ways of handling abrupt termination of `while` loops: *1*) syntactically, by enriching the logic with labeled modalities $[]_R$ and $\langle\rangle_R$, referring to the reason R of possible abrupt termination;

2) semantically, by transforming the program into an equivalent one which catches all top-level exceptions and thus always terminates normally. Parts of the programming logic were proved correct using an existing Isabelle/HOL formalization [86] of Java semantics. We prove the soundness of the method for loops with abrupt termination by transforming the loop into a normal terminating one. The transformation method looks similar to the one performed by KeY, however, we did not find any references on how the translation is performed, nor how the proof of the correctness was handled. Distinct to KeY system, our programming logic is first-order logic extended with meta-level constructs representing the program statements. In ESC/JAVA 2 [31], abruptly terminating loops can be modeled by `throw`/`catch` clauses. However, the system uses an unsound calculus; one source of unsoundness is the loop-unrolling mechanism rather than a loop invariant. Unlike this, our method is sound because we are using loop invariants to characterize loops.

In the LOOP tool [6], program semantics is described in type theory and has a memory model and semantics inheritance as basic ingredients. A Hoare-like calculus including abrupt termination is developed which is proved to be sound w.r.t. their approach to semantics. The correctness proofs of the calculus are done interactively in PVS [70] and Isabelle/HOL [67]. On the contrary, our approach does not need any memory model because we are working in a functional environment and the proofs are done automatically in the *Theorema* system.

The authors of [72] develop a structural operational semantics and Hoare-like logic as part of the Jive system [63]. The program logic is interactively proved sound w.r.t. the semantics by translating both of them in higher-order logic using Isabelle/HOL [67]. In comparison with the LOOP system, which reasons at semantic level, the reasoning of JIVE is at syntactic level. In contrast, our semantics is expressed in the logic on which the program operates and the correctness proof of the calculus as well, except some second order inferences.

A formalized semantics (in higher-order logic) of the C programming language is given in [68]. It handles the cases of abrupt termination by translating, at syntactic level, the abruptly terminating program into a normal terminating one and deriving post-conditions for each case of termination. The formalization is done in Isabelle/HOL [67].

The idea of using induction for termination proving has been widely used explicitly [69] or implicitly [18]. These proving techniques can be seen in the context of our work as methods for proving certain classes of inductive termination conditions that we generate. Note that in our approach termination is formulated as an induction principle and not used as a proving technique for termination as in the existing approaches.

Most of the proof assistants provide infrastructure for proving/disproving the termination of classical examples with general recursion. ACL2 [50] handles total functions that must be proved total at the definition time; sometimes the system is able to infer this fact. Isabelle [67], HOL4 [34], and Coq [7] are basically using the recursion package TFL [79] and thus allow definitions of total recursive functions by using the fixed-point operators and well-founded relations supplied by the user. Proving termination reduces to show that the relation is well-founded and the arguments of the recursive calls are decreasing. Our approach is equivalent, in the sense that the termination condition is equivalent to the well-foundedness of the partial order defined by the transformation of the critical variables[2] within the loop. The treatment of termination in [54] also uses inductive conditions extracted from the program recursions, but in the form of implicit definitions of domains (set theory is also needed). However, the existence of such inductively defined objects is not proved directly. Since our study is foundational, it constitutes a complement and not a competitor for practical work dealing with termination proofs, like e.g. termination of term rewriting systems`http://www.termination-portal.org/`, the size-change termination principle [57] or the approaches for proving the termination of industrial-size code (Microsoft Windows Operating SystemDrivers) [18].

In order to prove the correctness of the `while` loops in the classical Hoare logic it suffices to prove that the invariant holds upon loop execution and that a termination term exists. In case of abrupt terminating loops, one way is to introduce, at syntactic level, the notion of abnormal state [45] in the correctness statement. In these approaches the correctness proof was done by proving the correctness of the loop depending on the current statement which can occur in the loop body, therefore logical formulas and program statements appear in the proof. In our approach, we transform the loop into logical formulas and prove loop correctness based on them. Therefore the computed-supported proof in our case is simpler because it has to deal only with formulas from the logic on which the program operates.

Applying *computer algebra methods to program analysis and synthesis* is a challenging and relatively new research area. Challenging, because the polynomial algebra methods, although very powerful, suffer from high computational complexity. As a consequent, they often fail to be applied even to moderate sized programs. Successful applications are in the areas of:

- invariant generation, by combining methods like Gröbner bases, Cylindrical Algebraic Decomposition (CAD), symbolic summation, recurrence solving and generating functions [48, 53, 75]
- proving the correctness of imperative programs, by using Gröbner bases, CAD [21] or of hybrid systems [71].

However, we are not aware of related work combining QE based CAD with constraint optimization methods in order to synthesize numeric algorithms. Logical approaches to program synthesis are based on, e.g. induction, program schemes, model-checking, and have been successfully applied to the synthesis of decision pro-

[2] A critical variable is a program variable modified in the loop body.

cedures [47], Gröbner bases algorithm [12], synthesis of automata [14]. [80] presents a synthesis technique which is very much in the spirit of our work in the sense that annotations like invariant and termination term are used in the synthesis process. Contrary to our work, their technique has been used to synthesize a wide range of programs (integer square root, dynamic programming algorithms, sorting algorithms) using SMT solvers. However, the programs they synthesize have mainly linear expressions. The case study on integer square root they present contains only few quadratic expressions which can easily be handled once appropriate quadratic equalities/inequalities are fed as assumptions to the SMT solver. In this thesis, we focus on the synthesis of optimal algorithms for computing the square root of a real number. Computing the square root of a given real number is a fundamental operation. Naturally, various numerical methods have been developed [5, 15, 33, 37, 62, 65, 90]. We consider an interval version of the problem [1, 64, 73] and show how optimal algorithms can be synthesized under natural assumptions.

1.2. Contributions of the Thesis

We developed, formalized, and proved automatically, in the *Theorema* system, the soundness of a method for the verification of imperative recursive programs. The aim of the method is to identify the minimal logical apparatus necessary for formulating and proving (in a computer-assisted manner) a correct collection of methods for program verification. The study of such a minimal logical apparatus has the potential to increase the confidence in program verification tools and even to reveal some foundational relations between logic and programming. The distinctive features of our approach are:

- Program correctness is expressed in predicate logic, without using any additional theoretical model for program semantics or program execution, but only using the so-called object theories, theories relevant to the predicates, constants and functions present in the program text.
- The semantics of a loop is the implicit definition, at object level, of the function implemented by the loop.
- Termination is defined as an induction principle developed from the structure of the program with respect to `while` loops.

For proving the soundness, the entire knowledge base is formulated only in the logic on which the program operates except some axioms of natural number theory (including induction over natural numbers). Moreover, the proofs are performed using mainly first-order inferences (exception is Skolemization). We identified a reasonable-size knowledge base and a small set of inference rules which are handled efficiently during proof search by our predicate logic prover implemented in the *Theorema* system. Our computer-aided formalization may open the possibility of reflection of the method on itself (treatment of the meta-functions as programs whose correctness can be studied by the same method). Finally, the formal specification and the verification of the method are performed in the same framework, namely *Theorema* system. This facilitates reasoning at object and meta-level in the same system. Currently, our method can be applied to programs with single recursion and with arbitrarily-nested loops with abrupt termination (`break`, `return`).

Our results are as follows. We present the full formalization of our method in Section 2.2. In Section 2.3.1, we present the soundness of our approach for single recursive programs. Finally, in Sections 2.3.2 and 2.3.3, the method is proved to be sound for programs with arbitrarily-nested, abruptly terminating, `while` loops. We also investigated ways to synthesize reliable/optimal numeric algorithms. As a case study, we synthesized optimal algorithms for computing the square root of a real number. More precisely, given the real number x and the error bound ε, we are searching for a real interval such that it contains $\sqrt{x}$ and its width is less than ε, by using iterative refining algorithm scheme. Iterative refining means that the algorithm starts with an initial interval and repeatedly updates it by applying a refinement function, say f, on it until it becomes narrow enough. The synthesis amounts to finding a refinement function f that ensures that the algorithm is correct (loop invariant), terminating (contraction), and optimal. All these can be formulated as QE over the real numbers. Hence, in principle, they can be all carried out automatically. However, the computational requirement is so huge, making the automatic synthesis practically impossible with the state-of-the art QE software. Hence, we did some hand derivations and were able to synthesize semi-automatically optimal algorithms under suitable assumptions. Our first result (Section 3.4) consists in the proof that the well-known Secant-Newton function is the optimal among all its natural generalizations, that is, among functions which that are contracting and are quadratic rational functions in the end points of the input interval. Additionally, we proved that all natural generalizations of Secant-Newton function, including the function itself, have the best Lipschitz constant $\frac{1}{2}$ (Section 3.5). A Lipschitz constant strictly less than 1 ensures that the refinement function converges to $\sqrt{x}$ and how fast it does. By dropping off the contraction condition of the refinement function and imposing other natural constraints on it (Secant-Newton refining function satisfies these constraints), we were able to synthesize semi-automatically new refining functions for which the best Lipschitz constant is $\frac{1}{4}$ (Section 3.6). Hence, we synthesized faster convergent refinement functions than the well-known Secant-Newton.

1.3. Structure of the Thesis

Chapter 2 presents the verification method and the automated proof of soundness of it as follows. We start in Section 2.1 by motivating program analysis by symbolic execution and functional semantics which are the basic tools for developing the verification method. In Section 2.2 we describe a meta-theory for reasoning about imperative recursive programs. We consider the syntax, semantics and generation of the verification conditions and we exemplify these notions on several examples. In Section 2.3 we present the soundness proof of our approach on different types of programs: single recursive programs and programs with (nested, abruptly terminating) imperative loops. Section 2.4 starts with a brief presentation of the *Theorema* system, the tool we used for the automation and for the proof of soundness of the verification method. Then it continues with the description of: *i*) *Theorema* language layers and their usage in our research (Section 2.4.2), *ii*) *Predicate Logic Prover* and the extensions we performed in order to prove automatically the soundness of our verification method (Section 2.4.3), and *iii*) *FwdVCG*, the verification conditions generator which adds program analysis by symbolic execution and functional semantics feature to the *Theorema* system (Section 2.4.4).

Chapter 3 presents the results obtained in the synthesis of optimal real square root computation as follows. We describe in Section 3.1 how the program synthesis task can be reduced to a program verification task. We exemplify this transformation by giving a motivating example: real square root computation by Secant-Newton refinement function (algorithm). This example brings into attention the problem of synthesizing algorithms with a better complexity. We formulate the synthesis problem as a QE task over real numbers in Section 3.2 and we give an algorithm which, theoretically, could solve this problem. The synthesis algorithm uses CAD technique for quantifier elimination (Section 3.3). The input of the synthesis algorithm is so complex that it makes the QE infeasible with the available software CAD based software. Hence, we simplify it by imposing assumptions which exploit the deep knowledge on the problem. In this way we were able to prove semi-automatically that: *i*) Secant-Newton refinement map is optimal among all its natural generalizations, that is, among functions which are contracting and are quadratic rational functions in the end points of the input interval (Section 3.4) *ii*) all natural generalizations of Secant-Newton function, including the function itself, have the best Lipschitz constant $\frac{1}{2}$ (Section 3.5), and *iii*) by dropping off the contraction condition of the refinement function but imposing other natural assumptions, Secant-Newton refining function can be outperformed (Section 3.6).

In Chapter 4 we conclude and propose possible extensions of our work. Finally, the appendices present automatically generated *Theorema* proofs of the soundness of the verification method (Appendices A and B) and automatically obtained results of the synthesis problem (Appendix C) using the computer algebra system *Mathematica* [92].

2. Automated Static Analysis of Algorithms

2.1. Program Verification by Symbolic Execution

Symbolic execution has its origins back in 1976, when James King presented the method and the computer implementation of it in the EFFIGY system [51]. The technique replaces the input values of the variables by symbolic values and uses these new values to transform the program into first-order logical formulas (verification conditions) based on predicate transformers, which work either forward, or backward on the source code of the program. Because the input variables have symbolic values, the program variables have symbolic values in each state.

Two notions are involved in program verification using this approach: *the program state* and *the path condition*. The *program state* contains the values of the program variables and the statement counter. The *values of the program variables* are gathered in the program substitution σ, a set of replacements of the form $v \to e$ (v has the value e). The *program counter* determines which statement will be analyzed next. The *path condition* accumulates constraints which the inputs must satisfy such that the program execution follows the corresponding program branch.

For the purpose of generating the path conditions it turned out that *forward reasoning* [19] (used in the the majority of symbolic execution systems) is more suitable than *backward reasoning* [44], because it follows naturally the execution of a program.

Former symbolic execution systems (see [19] for a survey) were specialized in the generation of the verification conditions for each path of the program, detection of infeasible paths, computation of the output value of the programs in terms of the input values, etc. The last enumerated feature is called *functional semantics* in our approach.

In the last years, symbolic execution is used: *i*) for combating the state-space explosion problem in the model checking of programs that take input from unbounded domains with complex structure [2], *ii*) in separation logic [46], *iii*) in combination with other specification methodologies, e.g. dynamic logic [4], implicit dynamic frames [77].

2.2. Logical Foundations of Imperative Recursive Programs

The verification method developed by us is logic-based, meaning that the program correctness is provable in predicate logic, without using any additional theoretical model for program semantics or program execution, but only using the theories relevant to the predicates, constants and functions present in the program text. We call such theories *object theories*. (By a *theory* we understand a set of formulas in the language of predicate logic with equality.)

From the point of view of the program analysis, we distinguish the following types of functions:

- *basic* – occur in the object theory, have input condition, but no output condition; for instance, arithmetic operations in various number domains;
- *additional* – occur in the object theory, are usually functions implemented by other programs and in the process of verification conditions generation only their specification will be used.

A *meta-theory* (in predicate logic with equality) is further constructed for the purpose of reasoning about the correctness of programs. While the *object theory* is application specific, the *meta-theory* is universal. The meta-theory contains:

- specific functions and predicates from the set theory;
- elements from the tuple theory;
- function symbols for the construction of program statements (assignment including recursive call, conditionals, loops, abrupt statements: `break`, `return`);
- definitions of meta-predicates checking the syntactic correctness and of meta-functions defining the semantics and generating verification conditions.

A program P is a tuple of statements and is documented with specification: input $I_P[\alpha]$ and output $O_P[\alpha, \beta]$ condition. It takes as input a certain number of variables, conventionally denoted by α and it returns a single value, conventionally denoted by β. The program itself and program statements are meta-terms. Also the terms and the formulas from the object theory are meta-terms from the point of view of the meta-theory.

The expressions composing the definitions of the meta-level predicates and functions from the meta-theory are to be understood as universally quantified over the meta-variables of various types: $v \in V \subset \mathcal{V}$ is an initialized variable, $t \in \mathcal{T}$ is a term, φ is a boolean expression, B, P_T and P_F are tuples of statements representing the loop body and the two paths corresponding to the `if` statement, respectively. ι and ι' denote conventionally loop invariants which hold at the beginning of the loop, respectively, and are inductively preserved by each loop iteration. We assume that the loops are annotated with invariants. We denote conventionally by δ the critical variables, that is, the variables which are modified in the loop body.

The meta-predicates and meta-functions use forward symbolic execution in program analysis. How symbolic execution is used for different tasks (syntax checking, semantics construction, and generation of verification conditions) is presented in Sections 2.2.1, 2.2.2, and 2.2.3. Note that there is a predicate/function analyzing the main program, which calls auxiliary predicates/functions if `while` loops (with abrupt termination) are encountered. The definition of auxiliary predicates/functions could have been avoided by introducing global variables checking whether we are/we are not in a (nested) loop (with abrupt termination). However, we prefer specialized auxiliary predicate for the cleanliness of the formalization.

For the purpose of exemplification of our approach, we introduce the following algorithms. Algorithm 1 is a recursive algorithm computing the greatest common divisor

Algorithm 1 Greatest Common Divisor by Euclidean Algorithm

`in` a, b: `integers` where $a \geq 0$, $b \geq 0$
`out` β: `integer` where $\underset{k}{\exists}(a = k * \beta)$
`if` $(a = 0)$ `then`
    `return`$[b]$;
`if` $(b \neq 0)$ `then`
    `if` $(a > b)$ `then`
        $a := \mathrm{GCD}[a - b, b]$,
        $a := \mathrm{GCD}[a, b - a]$;
`return`$[a]$

of two positive integers by Euclidean algorithm. Because our aim is the exemplification of our approach, we simplified the postcondition of the algorithm: it is not the one of the greatest common divisor, but of a common divisor. Algorithm 2 presents a

Algorithm 2 Linear Search

`in` a: `array of integers`; n, e: `integers` where $n \geq 0$
`out` β: `boolean` where $\left(\underset{0 \leq j < n}{\forall} a[j] \neq e \wedge \beta = \mathbb{F}\right) \vee \left(\underset{0 \leq j < n}{\exists} a[j] = e \wedge \beta = \mathbb{T}\right)$
$i := 0$; $y := \mathbb{F}$;
`while` $(i < n)$ `do`
    `if` $(e = a[i])$ `then` $y := \mathbb{T}$; `break`;
    $i := i + 1$
`return`$[y]$

searching algorithm of the element e in the array a and returns $\mathbb{T}$ (true), if the element was found, or $\mathbb{F}$ (false), otherwise. The loop is manually annotated with the invariant

$$I[i, y] \;:\Longleftrightarrow\; 0 \leq i \leq n \wedge \left(\left(\underset{0 \leq j < i}{\forall} a[j] \neq e \wedge y = \mathbb{F}\right) \vee (a[i] = e \wedge y = \mathbb{T})\right)$$

Algorithm 3 Search in a bidimensional array

<u>in</u> a: <u>`array of integers`</u>; m, n, e: <u>`integers`</u> $m \geq 0$, $n \geq 0$
<u>out</u> β: <u>`integer`</u> or β_1, β_2: <u>`integers`</u> where
$\left(\underset{0\leq k<m}{\exists}\ \underset{0\leq l<n}{\exists} a[k][l] = e \wedge a[\beta_1][\beta_2] = e\right) \vee \left(\underset{0\leq k<m}{\forall}\ \underset{0\leq l<n}{\forall} a[k][l] \neq e \wedge \beta = -1\right)$
$i := 0$; $j := 0$;
<u>`while`</u> $(i < m)$ <u>`do`</u>
    $j := 0$;
    <u>`while`</u> $(j < n)$ <u>`do`</u>
        <u>`if`</u> $(e := a[i][j])$ <u>`then`</u> <u>`return`</u>$[\langle i, j\rangle]$;
        $j := j + 1$;
    $i := i + 1$;
<u>`return`</u>[-1]

Algorithm 3 searches for the element e into the bidimensional array a and returns its position if the element was found and -1, otherwise. We assume that the outer and inner loop are annotated with the invariants ι_1, respectively, ι_2.

2.2.1. Syntax and Semantics

We first introduce meta-predicates checking the syntactic correctness of programs and meta-functions which construct the program semantics. These are not actually needed for the implementation of a program verification system. They are only needed in order to reason about the effect of the verification condition generator. For instance, all statements about the effect of the meta-functions can be formulated only on programs which fulfill the syntax predicates. Likewise, the effect of a program P is expressed as a logical formula, which constitutes the implicit definition of the function realized by the program. Additionally, we construct the semantics of each loop as an implicit definition of the function implemented by the loop on the critical variables.

Syntax

The predicate Π checks that: *i*) a program is syntactically correct, *ii*) each variable is initialized before it is used, *iii*) each program branch has a <u>`return`</u> statement, and *iv*) <u>`break`</u> statement occurs only in <u>`while`</u> loops. The meta-level function $Vars$ constructs a list containing the variables occurring in a term or formula, and the meta-level predicate $IsFOLFormula$ checks whether an expression is a first-order logic formula. V is the set of initialized variables.

Definition 2.1.

1. $\Pi[P] :\Leftrightarrow \bigwedge \begin{cases} IsFOLFormula[I_P[\alpha]] \\ IsFOLFormula[O_P[\alpha, \beta]] \\ \Pi[\{\alpha \to \alpha_0\}, P]\{\alpha_0 \to \alpha\} \end{cases}$
2. $\Pi[V, \langle \underline{\texttt{return}}[t] \rangle \smile P] :\Leftrightarrow Vars[t] \subseteq V$
3. $\Pi[V, \langle v{:=}t \rangle \smile P] :\Leftrightarrow Vars[t] \subseteq V \wedge \Pi[V \cup \{v\}, P]$
4. $\Pi[V, \langle \underline{\texttt{if}}\ \varphi\ \underline{\texttt{then}}\ P_T, P_F \rangle \smile P] :\Leftrightarrow \bigwedge \begin{cases} Vars[\varphi] \subseteq V \\ IsFOLFormula[\varphi] \\ \Pi[V, P_T \smile P] \\ \Pi[V, P_F \smile P] \end{cases}$
5. $\Pi[V, \langle \underline{\texttt{while}}\ \varphi\ \underline{\texttt{do}}\ \iota,\ B \rangle \smile P] :\Leftrightarrow \bigwedge \begin{cases} Vars[\varphi] \subseteq V \\ IsFOLFormula[\varphi] \wedge IsFOLFormula[\iota] \\ \Pi'[V, B \smile P] \\ \Pi[V, P] \end{cases}$
6. $\Pi[V, \langle \underline{\texttt{assert}}[\varphi] \rangle \smile P] :\Leftrightarrow IsFOLFormula[\varphi] \wedge \Pi[V, P]$
7. $\Pi[V, P] = \mathbb{F}$

The input variable α from Definition 2.1.1 behaves like a global variable. Some of the principles of the syntactic check are as follows. The variables occurring in a term t or in a formula φ have to be initialized (Definition 2.1.2). The formulas I_P, O_P, φ, and ι must be well-formed first-order formulas (Definition 2.1.5). In case of successful assignment, the variable v is added to the set V of initialized variables. `break` statement outside the loop body gives a syntax error (Definition 2.1.7), however it is allowed inside the loop body (Definition 2.2.2). This is also the reason why the auxiliary predicate Π' was introduced: to make distinction among `break` behavior. Absence of `return` on each program path gives syntax error (Definition 2.1.7).

The meta-predicate Π', except for the `break` statement, behaves similarly to Π.

Definition 2.2.

1. $\Pi'[V, \langle \underline{\texttt{return}}[t] \rangle \smile P] :\Leftrightarrow Vars[t] \subseteq V$
2. $\Pi'[V, \langle \underline{\texttt{break}} \rangle \smile P] :\Leftrightarrow \mathbb{T}$
3. $\Pi'[V, \langle v{:=}t \rangle \smile P] :\Leftrightarrow Vars[t] \subseteq V \wedge \Pi'[V \cup \{v\}, P]$
4. $\Pi'[V, \langle \underline{\texttt{if}}\ \varphi\ \underline{\texttt{then}}\ P_T, P_F \rangle \smile P] :\Leftrightarrow \bigwedge \begin{cases} Vars[\varphi] \subseteq V \\ IsFOLFormula[\varphi] \\ \Pi'[V, P_T \smile P] \\ \Pi'[V, P_F \smile P] \end{cases}$
5. $\Pi'[V, \langle\rangle] :\Leftrightarrow \mathbb{T}$
6. $\Pi'[V, \langle \underline{\texttt{while}}\ \varphi\ \underline{\texttt{do}}\ \iota, B \rangle \smile P] :\Leftrightarrow \bigwedge \begin{cases} Vars[\varphi] \subseteq V \\ IsFOLFormula[\varphi] \wedge IsFOLFormula[\iota] \\ \Pi'[V, B \smile P] \\ \Pi'[V, P] \end{cases}$

7. $\Pi'[V, \langle \underline{\texttt{assert}}[\varphi] \rangle \smile P] :\Leftrightarrow IsFOLFormula[\varphi] \wedge \Pi'[V, P]$
8. $\Pi'[V, P] :\Leftrightarrow \mathbb{F}$

Semantics

We define the semantics of programs as an implicit definition at *object level* of the function implemented by the program. The semantics of a program is a formula with the shape:

$$\underset{\alpha:I_P}{\forall} \bigwedge_{i=1}^{n} \left(p_i[\alpha] \Rightarrow (\mathcal{F}[\alpha] = t_i) \right), \tag{2.1}$$

where $\mathcal{F}$ is a new (second order) symbol – a name for the function defined by the program, n is the number of paths in the program. In case of recursive calls, $\mathcal{F}$ may occur in some $p_i[\alpha]$ and t_i.

Each conjunct of (2.1) is a conditional definition for $\mathcal{F}[\alpha]$ which depends on the path condition p_i and on $\underline{\texttt{return}}$ statement of the respective path, whose argument (symbolically evaluated) represents the corresponding value of $\mathcal{F}[\alpha]$, namely t_i. For programs with loops, the behavior of a certain loop is not reflected explicitly at upper level (it is encoded into invariant), except for abrupt termination.

Formulas of type (2.1) are generated by the meta-level function Σ, Σ', and Σ''. The arguments of these functions are: *i*) substitution σ, *ii*) path condition Φ, *iii*) program counter, and *iv*) a name for the program/loop function. The output is a concatenation of tuples, each tuple having the form (2.1).

The main meta-level function Σ starts the analysis by assigning symbolic values to the input program variables and the input condition as path condition (Definition 2.3.1), and then updates the program substitution σ and the path condition according to the statements of the program.

Definition 2.3.

1. $\Sigma[P] = \Sigma[\{\alpha \to \alpha_0\}, I_P[\alpha_0], P, \mathcal{F}]\{\alpha_0 \to \alpha\}$
2. $\Sigma[\sigma, \Phi, \langle \underline{\texttt{return}}[t] \rangle \smile P, \mathcal{F}] = \langle \Phi \Rightarrow (\mathcal{F}[\alpha_0] = t\sigma) \rangle$
3. $\Sigma[\sigma, \Phi, \langle \underline{\texttt{break}} \rangle \smile P, \mathcal{F}] = \langle \rangle$
4. $\Sigma[\sigma, \Phi, \langle v{:=}t \rangle \smile P, \mathcal{F}] = \Sigma[\sigma\{v \to t\sigma\}, \Phi, P, \mathcal{F}]$
5. $\Sigma[\sigma, \Phi, \langle \underline{\texttt{if}}\ \varphi\ \underline{\texttt{then}}\ P_T, P_F \rangle \smile P, \mathcal{F}] = \smile \begin{cases} \Sigma[\sigma, \Phi \wedge \varphi\sigma, P_T \smile P, \mathcal{F}] \\ \Sigma[\sigma, \Phi \wedge \neg\varphi\sigma, P_F \smile P, \mathcal{F}] \end{cases}$
6. $\Sigma[\sigma, \Phi, \langle \rangle, \mathcal{F}] = \langle \rangle$
7. $\Sigma[\sigma, \Phi, \langle \underline{\texttt{while}}\ \varphi\ \underline{\texttt{do}}\ \iota\ B \rangle \smile P, \mathcal{F}] =$

$$\smile \left\{ \begin{array}{l} \Sigma[\sigma, \Phi \wedge \neg\varphi\sigma, P, \mathcal{F}] \quad (1) \\ \left\langle \underset{\delta:\iota}{\forall} \wedge \left\{ \begin{array}{l} (\neg\varphi\sigma_0 \Rightarrow (f[\delta] = \delta\sigma))\{\delta_0 \to \delta\} \\ \Sigma'[\sigma_0, \varphi\sigma_0, B, f]\{\delta_0 \to \delta\} \end{array} \right\} \right\rangle \quad (2) \\ \Sigma[\sigma_0, \Phi \wedge \varphi\sigma_0 \wedge \iota\sigma_0, B, \mathcal{F}] \quad (3) \\ \Sigma[\sigma_0, \Phi \wedge \neg\varphi\sigma_0 \wedge \iota\sigma_0, P, \mathcal{F}] \quad (4) \end{array} \right.$$

8. $\Sigma[\sigma, \Phi, \langle \underline{\texttt{assert}}[\varphi]\rangle \smile P, \mathcal{F}] = \Sigma[\sigma, \Phi \wedge \varphi\sigma, P, \mathcal{F}]$

A return statement constructs the expression of the program function on the respective program path (Definition 2.3.2), if forks the program execution (Definition 2.3.5). Definitions 2.3.3 (break) and 2.3.6 (end of the loop) are applied only in the case the currently analyzed module[1] has no nested loops. Assignment of a term (including recursive call) updates the program substitution (Definition 2.3.4). Semantics of programs with while loops (Definition 2.3.7) is constructed as follows. Definitions 2.3.7.1, 2.3.7.3 and 2.3.7.4 construct the semantics of the main program, in particular Definition 2.3.7.3 searches for program branches with abrupt termination. Definition 2.3.7.2 constructs the semantics of the loop. If a loop abruptly terminates via break, then specialized semantics definition exist due to distinct outcome of break inside the loop (Definition 2.4.2), respectively outside (Definition 2.5.2). Note that the analysis of the loop starts with fresh values for the critical variables, fact denoted by the substitution σ_0. An assert construct (Definition 2.3.8) updates the path condition and afterwards continues the analysis of the program.

Definition 2.4.

1. $\Sigma'[\sigma, \Phi, \langle \underline{\texttt{return}}[t]\rangle \smile P, f] = \big(\Phi \Rightarrow (f[\delta_0] = t\sigma)\big)$
2. $\Sigma'[\sigma, \Phi, \langle \underline{\texttt{break}}\rangle \smile P, f] = \big(\Phi \Rightarrow (f[\delta_0] = \delta_0\sigma)\big)$
3. $\Sigma'[\sigma, \Phi, \langle v{:=}t\rangle \smile P, f] = \Sigma'[\sigma\{v \to t\sigma\}, \Phi, P, f]$
4. $\Sigma'[\sigma, \Phi, \langle \underline{\texttt{if}}\ \varphi\ \underline{\texttt{then}}\ P_T, P_F\rangle \smile P, f] = \wedge \left\{ \begin{array}{l} \Sigma'[\sigma, \Phi, P_T \smile P, f] \\ \Sigma'[\sigma, \Phi, P_F \smile P, f] \end{array} \right.$
5. $\Sigma'[\sigma, \Phi, \langle\rangle, f] = \big(\Phi \Rightarrow (f[\delta_0] = f[\delta\sigma])\big)$
6. $\Sigma'[\sigma, \Phi, \langle \underline{\texttt{while}}\ \varphi\ \underline{\texttt{do}}\ \iota B\rangle \smile P, f] = \Sigma''[\sigma, \Phi, \langle \underline{\texttt{while}}\ \varphi\ \underline{\texttt{do}}\ \iota B\rangle \smile P, f]\{\delta_0 \to \delta\}$
7. $\Sigma'[\sigma, \Phi, \langle \underline{\texttt{assert}}[\varphi]\rangle \smile P, f] = \Sigma[\sigma, \Phi \wedge \varphi\sigma, P, f]$

Definition 2.5.

1. $\Sigma''[\sigma, \Phi, \langle \underline{\texttt{return}}[t]\rangle \smile P, f] = \big(\Phi \Rightarrow (f[\delta_0] = t\sigma)\big)$
2. $\Sigma''[\sigma, \Phi, \langle \underline{\texttt{break}}\rangle \smile P, f] = \mathbb{T}$
3. $\Sigma''[\sigma, \Phi, \langle v{:=}t\rangle \smile P, f] = \Sigma''[\sigma\{v \to t\sigma\}, \Phi, P, f]$
4. $\Sigma''[\sigma, \Phi, \langle \underline{\texttt{if}}\ \varphi\ \underline{\texttt{then}}\ P_T, P_F\rangle \smile P, f] = \wedge \left\{ \begin{array}{l} \Sigma''[\sigma, \Phi \wedge \varphi\sigma, P_T \smile P, f] \\ \Sigma''[\sigma, \Phi \wedge \neg\varphi\sigma, P_F \smile P, f] \end{array} \right.$
5. $\Sigma''[\sigma, \Phi, \langle\rangle, f] = \mathbb{T}$

[1] We call *module* a program or a loop.

6. $\Sigma''[\sigma, \Phi, \langle \underline{\texttt{while}}\ \varphi\ \underline{\texttt{do}}\ \iota B\rangle \smile P, f] = \wedge \begin{cases} \Sigma'[\sigma, \Phi \wedge \neg\varphi\sigma, P, f] \\ \Sigma''[\sigma_0, \Phi \wedge \varphi\sigma_0 \wedge \iota\sigma_0, B, f] \\ \Sigma'[\sigma_0, \Phi \wedge \neg\varphi\sigma_0 \wedge \iota\sigma_0, P, f] \end{cases}$

7. $\Sigma''[\sigma, \Phi, \langle \underline{\texttt{assert}}[\varphi]\rangle \smile P, f] = \Sigma''[\sigma, \Phi \wedge \varphi\sigma, P, f]$

Remark 2.6. The functions ensure that all program branches are analyzed and the path conditions are mutually disjoint.

Remark 2.7. The functions translate the program into a *function*. From this point on, one could reason about the program using the Scott fixpoint theory [58, p.86], however we prefer a logic-based approach.

For example, the semantics of Algorithm 1 is as follows. (The numbers in parentheses represent program lines.)

$$\underset{a \geq 0 \wedge b \geq 0}{\forall} \bigwedge \begin{cases} a = 0 \Rightarrow (\mathrm{GCD}[a, b] = b) & (1, 3, 4) \\ (a \neq 0 \wedge b \neq 0 \wedge a > b) \Rightarrow (\mathrm{GCD}[a, b] = \mathrm{GCD}[a - b, b]) & (1, 3, 5, 6, 7, 9) \\ (a \neq 0 \wedge b \neq 0 \wedge a \leq b) \Rightarrow (\mathrm{GCD}[a, b] = \mathrm{GCD}[a, b - a]) & (1, 3, 5, 6, 8, 9) \\ a \neq 0 \wedge b \neq 0 \Rightarrow (\mathrm{GCD}[a, b] = a) & (1, 3, 5, 9) \end{cases} \tag{2.2}$$

Formula (2.2) states the following: "For all values of the input variables a and b satisfying the input condition $a \geq 0 \wedge b \geq 0$, on the path where: *i)* $a = 0$, the value of the semantics function GCD is b, *ii)* $a \neq 0 \wedge b \neq 0 \wedge a > b$, the value of the semantics function GCD is computed recursively, with the value $a - b$ for the argument a and b remains unchanged, *iii)* $a \neq 0 \wedge b \neq 0 \wedge a \leq b$, the value of the semantics function GCD is computed recursively, with the value $b - a$ for the argument b and a remains unchanged, *iv)* $a \neq 0 \wedge b \neq 0$, the value of the semantics function GCD is a.

Algorithm 3 has two nested loops with abrupt termination. Semantics functions for the main program, inner and outer loops are generated. ι_1 and ι_2 are the loop invariants of the the outer, respectively, inner loop.

Semantics of the program.

$$\underset{m \geq 0 \wedge n \geq 0}{\forall} \bigwedge \begin{cases} 0 \geq m \Rightarrow (\mathcal{F}[m, n] = -1) & (1, 3, 4, 10) \\ i < m \wedge \iota_1 \wedge j < n \wedge \iota_2 \wedge (e = a[i, j]) \Rightarrow (\mathcal{F}[m, n] = \langle i, j\rangle) & (1, 4, 6, 7) \\ i \geq m \wedge \iota_1 \Rightarrow (\mathcal{F}[m, n] = -1) & (1, 4, 10) \end{cases}$$

Semantics of the outer loop.

$$\underset{i, j : \iota_1}{\forall} \bigwedge \begin{cases} i \geq m \Rightarrow (f_1[i, j] = \langle i, j\rangle) & (4) \\ i < m \wedge j \geq n \Rightarrow (f_1[i, j] = f_1[i + 1, j]) & (4, 6, 8) \\ i < m \wedge j < n \wedge \iota_2 \wedge (e = a[i, j]) \Rightarrow (f_1[i, j] = \langle i, j\rangle) & (4, 6, 7) \\ i < m \wedge j \geq n \wedge \iota_2 \Rightarrow (f_1[i, j] = f_1[i + 1, j]) & (4, 6, 8) \end{cases}$$

Semantics of the inner loop.

$$\underset{j:\iota_2}{\forall} \bigwedge \begin{cases} j \geq n \Rightarrow (f_2[j] = j) & (6) \\ j < n \wedge (e = a[i,j]) \Rightarrow (f_2[j] = \langle i, j \rangle) & (6,7) \\ j < n \wedge (e \neq a[i,j]) \Rightarrow (f_2[j] = f_2[j+1]) & (6,7,8) \end{cases}$$

2.2.2. Partial Correctness

Partial correctness verification conditions ensure *safety* and *functional correctness* of a program.

Safety conditions are formulas with the shape $\Phi \Rightarrow I_h[t]$, where Φ represents the path condition, I_h is the input condition of some function h, and t is the symbolic value of the argument of the function call. We require that all functions present in a program satisfy their input condition. This is not necessarily needed for partial correctness, however for practical programming is an important requirement.

An example of safety condition is:

$$a \geq 0 \wedge b \geq 0 \wedge a \neq 0 \wedge b \neq 0 \wedge a > b \implies a - b \geq 0 \wedge b \geq 0, \tag{2.3}$$

obtained by analyzing the lines $(1, 3, 5, 6, 7)$ of the Algorithm 1. Next, the program analysis proceeds by adding the condition $a - b \geq 0 \wedge b \geq 0$ ($I_{\mathrm{GCD}}[a, b]$) to the path condition. This is not actually necessary (because of (2.3)), however it might help in the proving process of (2.4) since there are more assumptions.

$$a \geq 0 \wedge b \geq 0 \wedge a \neq 0 \wedge b \neq 0 \wedge a > b \wedge a - b \geq 0 \wedge b \geq 0 \wedge \underset{k}{\exists}\, b = k \cdot t_1 \implies \underset{k}{\exists}\, a = k \cdot t_1. \tag{2.4}$$

Functional, respectively, *assertive conditions* are formulas checking that the output condition on the currently returned value, respectively, the assertion at a certain point in the program, is a consequence of the accumulated conditions on the respective program path. An example of functional verification condition is the formula:

$$m \geq 0 \wedge n \geq 0 \wedge 0 \geq m \Rightarrow$$
$$\left(\underset{0 \leq k < m}{\exists} \underset{0 \leq l < n}{\exists} a[k][l] = e \wedge a[\beta_1][\beta_2] = e \right) \vee \left(\underset{0 \leq k < m}{\forall} \underset{0 \leq l < n}{\forall} a[k][l] \neq e \wedge 1 = -1 \right),$$

obtained from the analysis of the program lines $(1, 3, 4, 10, 2)$ of Algorithm 3.

The partial correctness verification conditions are generated by the meta-level functions Γ and Γ'. The verification of the program is performed with respect to a given specification, whose definition is assumed to be present in the object theory. Moreover, the basic functions from the object theory have only input condition, but no output condition. These functions will occur in the verification conditions, thus the proof of such conditions will use the properties of the basic functions from the object theory. Typical examples of basic functions are the arithmetic operations in various number domains. The additional functions also have a specification. A certain additional function, say h, has the input condition $I_h[t]$ and the output condition $O_h[t,y]$, where y is a new symbol name which will be used subsequently as the output value of h. In this way, the verification conditions use only the specification, thus leaving room for possible changes in their implementation. For the particular case of recursive call, this technique is mandatory because the existence of the function implemented by the program is not automatically ensured.

The arguments of the functions Γ and Γ' are: *i)* program substitution σ, *ii)* path condition Φ, and *iii)* the program counter. The output is a list of first-order logic formulas (verification conditions). Similarly to the main semantics function, Γ starts with symbolic value for the input program variable and with the input condition as path condition and then updates them according to the statements of the program.

Definition 2.8.

1. $\Gamma[P] = \Gamma[\{\alpha \to \alpha_0\}, I_P[\alpha_0], P]\{\alpha_0 \to \alpha\}$
2. $\Gamma[\sigma, \Phi, \langle \underline{\texttt{return}}[t]\rangle \smile P] = \big\langle \Phi \Rightarrow O_P[\alpha_0, t\sigma]\big\rangle$
3. $\Gamma[\sigma, \Phi, \langle v{:=}t\rangle \smile P] = f\Gamma[\sigma\{v \to t\sigma\}, \Phi, P]$
4. $\Gamma[\sigma, \Phi, \langle v{:=}h[\alpha]\rangle \smile P] = \bigwedge \begin{cases} \Phi \Rightarrow I_h[\alpha\sigma] \\ \Gamma[\sigma \circ \{v \to h[\alpha\sigma]\}, \Phi \wedge I_h[\alpha\sigma], P] \end{cases}$
5. $\Gamma[\sigma, \Phi, \langle v{:=}g[\alpha]\rangle \smile P] = \bigwedge \begin{cases} \Phi \Rightarrow I_g[\alpha\sigma] \\ \Gamma[\sigma \circ \{v \to c\}, \Phi \wedge I_g[\alpha\sigma] \wedge O_g[\alpha\sigma, c], P] \end{cases}$
6. $\Gamma[\sigma, \Phi, \langle \underline{\texttt{if}}\ \varphi\ \underline{\texttt{then}}\ P_T, P_F\rangle \smile P] = \smile \begin{cases} \Gamma[\sigma, \Phi \wedge \varphi\sigma, P_T \smile P] \\ \Gamma[\sigma, \Phi \wedge \neg\varphi\sigma, P_F \smile P] \end{cases}$
7. $\Gamma[\sigma, \Phi, \langle\rangle] = \big\langle \Phi \Rightarrow \iota\sigma \big\rangle$
8. $\Gamma[\sigma, \Phi, \langle \underline{\texttt{while}}\ \varphi\ \underline{\texttt{do}}\ \iota\ B\rangle \smile P] = \smile \begin{cases} \Gamma[\sigma, \Phi \wedge \neg\varphi\sigma, P] \quad (1) \\ \big\langle \Phi \Rightarrow \iota\sigma \big\rangle \quad (2) \\ \Gamma'[\sigma_0, \iota\sigma_0 \wedge \varphi\sigma_0, \iota, B]\{\delta_0 \to \delta\} \quad (3) \\ \Gamma[\sigma_0, \iota\sigma_0 \wedge \neg\varphi\sigma_0, P] \quad (4) \end{cases}$
9. $\Gamma[\sigma, \Phi, \langle \underline{\texttt{assert}}[\varphi]\rangle \smile P] = \smile \begin{cases} \big\langle \Phi \Rightarrow \varphi\sigma \big\rangle \\ \Gamma[\sigma, \Phi \wedge \varphi\sigma, P] \end{cases}$

The verification conditions for partial correctness are generated as follows. `return` statement determines the generation of a functional verification condition for the respective program path (Definition 2.8.2), `if` forks the program analysis (Definition 2.8.6) and updates path condition correspondingly. Distinction has to be made for different types of assignments. If the assigned term is a basic function, say h, (Definition 2.8.4), additional or recursive function, say g, (Definition 2.8.5) then a safety verification condition is generated and analysis proceeds with updated program substitution and path condition, depending on the function type. If the assignment is a simple term 2.8.5 then just the program substitution is updated. `while` loops split the analysis of the program in three branches: one branch considers that the loop is not executed (Definition 2.8.8.1), another analyzes the body of the loop (Definition 2.8.8.3) ensuring that invariant is inductively preserved (Definition 2.8.8.2), the third one analyzes the rest of the program taking into account that the loop was executed (the invariant and the negated loop condition are added to the path condition) and terminates (Definition 2.8.8.4). An assert statement determines the generation of an assertive condition. Note that due to the syntax check, which does not allow a break in the main program, no corresponding inductive definition is needed.

The auxiliary function Γ' has two additional arguments, namely the loop condition and invariant. They were introduced for a correct formulation of Definition 2.9.8: the loop condition φ and invariant ι occur on the right hand side of the definition.

Definition 2.9.

1. $\Gamma'[\sigma, \Phi, \iota, \varphi, \langle \underline{\texttt{return}}[t]\rangle \smile P] = \left\langle \Phi \Rightarrow O_P[\alpha_0, t\sigma]\right\rangle$
2. $\Gamma'[\sigma, \Phi, \iota, \varphi, \langle \underline{\texttt{break}}\rangle \smile B \smile \langle\rangle \smile P] = \Gamma'[\sigma, \Phi, \iota, \varphi, P]$
3. $\Gamma'[\sigma, \Phi, \iota, \varphi, \langle v{:=}t\rangle \smile P] = \Gamma'[\sigma\{v \to t\sigma\}, \Phi, \iota, \varphi, P]$
4. $\Gamma'[\sigma, \Phi, \langle v{:=}h[\alpha]\rangle \smile P] = \bigwedge \begin{cases} \Phi \Rightarrow I_h[\alpha\sigma] \\ \Gamma'[\sigma \circ \{v \to h[\alpha\sigma]\}, \Phi \wedge I_h[\alpha\sigma], P] \end{cases}$
5. $\Gamma'[\sigma, \Phi, \langle v{:=}g[\alpha]\rangle \smile P] = \bigwedge \begin{cases} \Phi \Rightarrow I_g[\alpha\sigma] \\ \Gamma'[\sigma \circ \{v \to c\}, \Phi \wedge I_g[\alpha\sigma] \wedge O_g[\alpha\sigma, c], P] \end{cases}$
6. $\Gamma'[\sigma, \Phi, \iota, \varphi, \langle \underline{\texttt{if}}\ \varphi\ \underline{\texttt{then}}\ P_T, P_F\rangle \smile P] = \smile \begin{cases} \Gamma'[\sigma, \Phi \wedge \varphi\sigma, \iota, \varphi, P_T \smile P] \\ \Gamma'[\sigma, \Phi \wedge \neg\varphi\sigma, \iota, \varphi, P_F \smile P] \end{cases}$
7. $\Gamma'[\sigma, \Phi, \iota, \varphi, \langle\rangle] = \left\langle \Phi \Rightarrow \iota\sigma \right\rangle$
8. $\Gamma'[\sigma, \Phi, \iota, \varphi, \langle \underline{\texttt{while}}\ \varphi\ \underline{\texttt{do}}\ \iota'\ B\rangle \smile P] = \smile \begin{cases} \Gamma'[\sigma, \Phi \wedge \neg\varphi\sigma, \iota, \varphi, P] \\ \left\langle \Phi \Rightarrow \iota\sigma \right\rangle \\ \Gamma'[\sigma_0, \iota'\sigma_0 \wedge \varphi\sigma_0, \iota_1, \varphi, B]\{\delta_0 \to \delta\} \\ \Gamma'[\sigma_0, \iota_1\sigma_0 \wedge \neg\varphi\sigma_0, \iota_1, \varphi, P] \end{cases}$
9. $\Gamma'[\sigma, \Phi, \iota, \varphi, \langle \underline{\texttt{assert}}[\vartheta]\rangle \smile P] = \smile \begin{cases} \left\langle \Phi \Rightarrow \vartheta\sigma \right\rangle \\ \Gamma'[\sigma, \Phi \wedge \vartheta\sigma, \iota, \varphi, P] \end{cases}$

Additional to Γ, Γ' has an inductive definition also for break. When a break statement is encountered (Definition 2.9.2), the analysis of the current loop is left and continued with the analysis of the statements after the loop body, without resuming the configuration of program substitution and path condition.

2.2.3. Termination

We approach termination by generating a termination condition for each iterative structure of the program.

For instance, the termination condition for Algorithm 1 is:

$$\underset{a\geq 0\wedge b\geq 0}{\forall} \bigwedge \begin{cases} a=0 \Rightarrow \pi[a,b] & (1,3,4) \\ a\neq 0 \wedge b\neq 0 \wedge a>b \wedge \pi[a-b,b] \Rightarrow \pi[a,b] & (1,3,5,6,7,9) \\ a\neq 0 \wedge b\neq 0 \wedge a\leq b \wedge \pi[a,b-a] \Rightarrow \pi[a,b] & (1,3,5,6,8,9) \\ a\neq 0 \wedge b=0 \Rightarrow \pi[a,b] & (1,3,5,9) \end{cases} \Rightarrow \underset{a\geq 0\wedge b\geq 0}{\forall} \pi[a,b] \tag{2.5}$$

In formula (2.5), π is a new constant symbol, thus it behaves like a universally quantified predicate. This is why this formula is in fact an induction principle. The formula consists in an implication between two universally quantified parts, both over the input variables a and b satisfying the input condition. The left-hand side is a conjunction of implicational clauses, one for each path of the program.

The rationale behind (2.5) is as follows. Let us consider the predicate $\tau[a,b]$: "the loop terminates on the input a,b", whose definition is actually not known. The left-hand side of the implication represents a property $T[\pi]$ which should be fulfilled by the predicate τ. Intuitively, this property states that the program terminates if the condition $a = 0$, respectively, $a \neq 0 \wedge b = 0$, holds, and furthermore, corresponding to each recursive path, it states that the loop terminates on a,b if it terminates on the values of the recursive call. Intuitively, we consider that the predicate expressing termination is the strongest predicate obeying this property T. The termination condition states that the input condition is stronger than any predicate fulfilling T – thus it will be also stronger than τ. In this way we can express termination without explicit use of τ. This is, however, only an intuitive explanation, and in Section 2.3 we show rigourously that the termination condition is sufficient for the existence and uniqueness of the function implemented by the program.

Formula 2.5 was generated by the meta-function Θ. If the program contains (nested) loops then, additionally, meta-functions Θ' and Θ'' are applied.

The meta-functions $\Theta, \Theta', \Theta''$ follow also the principles of symbolic execution. The meta-level function Θ analyzes the current module and specializes itself to Θ' for the analysis of loops and to Θ'' for modules which contain nested loops, because the `break` statement has different behavior for nested, respectively non-nested loops. The arguments of these functions are: substitution σ, path condition Φ, program counter, and a name for the termination predicate of the program or loop function. The output is a list of formulas of type (2.6), one for each iterative structure of the program.

$$\left(\underset{\alpha:I_P}{\forall} \bigwedge_{i=1}^{n} (p_i[\alpha] \Rightarrow \pi[\alpha]) \right) \Rightarrow \underset{\alpha:I_P}{\forall} \pi[\alpha], \tag{2.6}$$

In (2.6) π is a constant symbol. In the case of iterative structures, $\pi[\alpha]$ may occur in some $p_i[\alpha]$. n is the number of paths of the program.

The meta functions inspect all program branches and collect `if` and `while` conditions, loop invariants, etc. Moreover, they collect the characterizations by output conditions of the values produced by calls to additional functions (Definition 2.10.5), including the currently defined recursive call. However, in the last case, one also collects the condition $\pi[\alpha\sigma]$ – that is the arbitrary predicate applied to the current symbolic values of the arguments of the recursive call (Definition 2.10.6). On each program branch, the collected conditions are used as premise of π, and then the conjunction of all these clauses (after reverting to free variables) is universally quantified over the input condition and is used as a premise in the final formula.

Each time a loop is encountered, a new symbol π standing for an arbitrary predicate is generated and the generation of the termination condition proceeds as follows. A termination condition for the currently analyzed loop is generated (Definition 2.10.9.1). A path analyzes the loop body searching for abrupt termination (Definition 2.10.9.2). The last program branch continues with the analysis of the statements after the loop (Definition 2.10.9.3). Note that same analysis is performed to each loop, independently of the degree of nestedness, due to Definition 2.10.9.2.

Definition 2.10.

1. $\Theta[P] = \Theta[\{\alpha \to \alpha_0\}, I_P[\alpha_0], P]\{\alpha_0 \to \alpha\}$

2. $\Theta[\sigma, \Phi, \langle \underline{\texttt{return}}[t]\rangle \smile P] = \langle\rangle$

3. $\Theta[\sigma, \Phi, \langle \underline{\texttt{break}}\rangle \smile P] = \langle\rangle$

4. $\Theta[\sigma, \Phi, \langle v{:=}t\rangle \smile P] = \Theta[\sigma\{v \to t\sigma\}, \Phi, P]$

5. $\Theta[\sigma, \Phi, \langle v{:=}h[\alpha]\rangle \smile P] = \Theta[\sigma\{v \to y\}, \Phi \wedge O_h[\alpha\sigma, y], P]$

6. $\Theta[\sigma, \Phi, \langle v{:=}g[\alpha]\rangle \smile P] = \Theta[\sigma\{v \to y\}, \Phi \wedge O_g[\alpha\sigma, y] \wedge \pi[\alpha\sigma], P]$

7. $\Theta[\sigma, \Phi, \langle \underline{\texttt{if}}\ \varphi\ \underline{\texttt{then}}\ P_T, P_F\rangle \smile P] = \smile \begin{cases} \Theta[\sigma, \Phi \wedge \varphi\sigma, P_T \smile P] \\ \Theta[\sigma, \Phi \wedge \neg\varphi\sigma, P_F \smile P] \end{cases}$

8. $\Theta[\sigma, \Phi, \langle\rangle] = \langle\rangle$

9. $\Theta[\sigma, \Phi, \langle \underline{\texttt{while}}\ \varphi\ \underline{\texttt{do}}\ \iota\ B\rangle \smile P] =$

$$\smile \begin{cases} \Big\langle \underset{\delta:\iota}{\forall} \wedge \left\{ \begin{array}{l} \big(\neg\varphi\sigma_0 \Rightarrow \pi[\delta]\big)\{\delta_0 \to \delta\} \\ \Theta'[\sigma_0, \varphi\sigma_0, B, \pi]\{\delta_0 \to \delta\} \Rightarrow \pi[\delta] \end{array} \right\} \Rightarrow \underset{\delta:\iota}{\forall} \pi[\delta] \Big\rangle \quad (1) \\ \Theta[\sigma_0, \varphi\sigma_0 \wedge \iota\sigma_0, B] \quad (2) \\ \Theta[\sigma_0, \mathbb{T}, P] \quad (3) \end{cases}$$

10. $\Theta[\sigma, \Phi, \langle \underline{\texttt{assert}}[\varphi]\rangle \smile P] = \Theta[\sigma, \Phi \wedge \varphi\sigma, P]$

The auxiliary functions Θ' and Θ'' behave similarly to Θ, except that:

1. They generate a disjunction of formulas (for the simplicity of the approach), one for each path analyzed, from which the termination of the loop must follow (Definition 2.10.9.1),
2. <u>`return`</u> has the same behavior in non- and nested loops: they return the accumulated path conditions (Definitions 2.11.1 and 2.12.1);
3. <u>`break`</u> behaves similarly to <u>`return`</u> in non-nested loops (Definition 2.11.1), but for programs with nested loops the analysis performed in inner loops is not visible in the wrapper ones (Definitions 2.12.2).
4. At the end of the non-nested loop, a path condition involving the termination

predicate π is constructed (Definition 2.11.7), while the analysis performed in the nested loops is not visible in the outer loops (Definition 2.12.7)

5. Nested loops are always analyzed by the meta-function Θ'' (Definition 2.11.8).

Definition 2.11.

1. $\Theta'[\sigma, \Phi, \langle \underline{\texttt{return}}[\delta]\rangle \smile P, \pi] = \Phi$
2. $\Theta'[\sigma, \Phi, \langle \underline{\texttt{break}}\rangle \smile P, \pi] = \Phi$
3. $\Theta'[\sigma, \Phi, \langle v{:=}t\rangle \smile P, \pi] = \Theta'[\sigma\{v \to t\sigma\}, \Phi, P, \pi]$
4. $\Theta[\sigma, \Phi, \langle v{:=}h[\alpha]\rangle \smile P] = \Theta[\sigma\{v \to y\}, \Phi \wedge O_h[\alpha\sigma, y], P]$
5. $\Theta[\sigma, \Phi, \langle v{:=}g[\alpha]\rangle \smile P] = \Theta[\sigma\{v \to y\}, \Phi \wedge O_g[\alpha\sigma, y] \wedge \pi[\alpha\sigma], P]$
6. $\Theta'[\sigma, \Phi, \langle \underline{\texttt{if}}\ \varphi\ \underline{\texttt{then}}\ P_T, P_F\rangle \smile P, \pi] = \vee \begin{cases} \Theta'[\sigma, \Phi \wedge \varphi\sigma, P_T \smile P, \pi] \\ \Theta'[\sigma, \Phi \wedge \neg\varphi\sigma, P_F \smile P, \pi] \end{cases}$
7. $\Theta'[\sigma, \Phi, \langle\rangle, \pi] = (\Phi \wedge \pi[\delta\sigma])$
8. $\Theta'[\sigma, \Phi, \langle \underline{\texttt{while}}\ \varphi\ \underline{\texttt{do}}\ \iota\ B\ \rangle \smile P, \pi] = \Theta''[\sigma, \Phi, \langle \underline{\texttt{while}}\ \varphi\ \underline{\texttt{do}}\ \iota\ B\rangle \smile P, \pi]\{\delta_0 \to \delta\}$
9. $\Theta'[\sigma, \Phi, \langle \underline{\texttt{assert}}[\varphi]\rangle \smile P, \pi] = \Theta'[\sigma, \Phi \wedge \varphi\sigma, P, \pi]$

Definition 2.12.

1. $\Theta''[\sigma, \Phi, \langle \underline{\texttt{return}}[\delta]\rangle \smile P, \pi] = \Phi$
2. $\Theta''[\sigma, \Phi, \langle \underline{\texttt{break}}\rangle \smile P, \pi] = \mathbb{F}$
3. $\Theta''[\sigma, \Phi, \langle v{:=}t\rangle \smile P, \pi] = \Theta''[\sigma\{v \to t\sigma\}, \Phi, P, \pi]$
4. $\Theta[\sigma, \Phi, \langle v{:=}h[\alpha]\rangle \smile P] = \Theta[\sigma\{v \to y\}, \Phi \wedge O_h[\alpha\sigma, y], P]$
5. $\Theta[\sigma, \Phi, \langle v{:=}g[\alpha]\rangle \smile P] = \Theta[\sigma\{v \to y\}, \Phi \wedge O_g[\alpha\sigma, y] \wedge \pi[\alpha\sigma], P]$
6. $\Theta''[\sigma, \Phi, \langle \underline{\texttt{if}}\ \varphi\ \underline{\texttt{then}}\ P_T, P_F\rangle \smile P, \pi] = \vee \begin{cases} \Theta''[\sigma, \Phi \wedge \varphi\sigma, P_T \smile P, \pi] \\ \Theta''[\sigma, \Phi \wedge \neg\varphi\sigma, P_F \smile P, \pi] \end{cases}$
7. $\Theta''[\sigma, \Phi, \langle\rangle, \pi] = \mathbb{F}$
8. $\Theta''[\sigma, \Phi, \langle \underline{\texttt{while}}\ \varphi\ \underline{\texttt{do}}\ \iota\ B\rangle \smile P, \pi] = \vee \begin{cases} \Theta'[\sigma, \Phi \wedge \neg\varphi\sigma, P, \pi] \\ \Theta''[\sigma_0, \varphi\sigma_0 \wedge \iota\sigma_0, B, \pi] \\ \Theta'[\sigma_0, \neg\varphi\sigma_0 \wedge \iota\sigma_0, P, \pi] \end{cases}$
9. $\Theta''[\sigma, \Phi, \langle \underline{\texttt{assert}}[\varphi]\rangle \smile P, \pi] = \Theta''[\sigma, \Phi \wedge \varphi\sigma, P, \pi]$

For Algorithm 3, two termination conditions are generated, one for each loop in the program. There are actually two induction principles, developed from the structure of the loops. The algorithm terminates if both loops terminates, i.e. the following two formulas hold.

Termination of the outer loop.

$$\underset{i,j:\iota_1}{\forall} \bigwedge \begin{cases} i \geq m \Rightarrow \pi_1[i,j] & (5,6) \\ i < m \wedge j \geq n \wedge \pi_1[i+1,j] \Rightarrow \pi_1[i,j] & (5,6,8,13) \\ i < m \wedge j < n \wedge \iota_2 \wedge (a[i,j] = e) \Rightarrow \pi_1[i,j] & (5,6,8,9,10) \\ i < m \wedge j \geq n \wedge \iota_2 \wedge \pi_1[i+1,j] \Rightarrow \pi_1[i,j] & (5,6,8,9,13) \end{cases} \Rightarrow \underset{i,j:\iota_1}{\forall} \pi_1[i,j]$$

Termination of the inner loop.

$$\underset{j:\iota_2}{\forall} \bigwedge \begin{cases} j \geq n \Rightarrow \pi_2[j] \\ j < n \wedge (a[i,j] = e) \Rightarrow \pi_2[j] \\ j < n \wedge (a[i,j] \neq e) \wedge \pi_2[j+1] \Rightarrow \pi_2[j] \end{cases} \Rightarrow \underset{j:\iota_2}{\forall} \pi_2[j] \qquad \begin{matrix} (8,9) \\ (8,9,10) \\ (8,9,10,12) \end{matrix}$$

2.3. Soundness of the Method

In order to perform automatically the soundness proof of our method for program verification, we extended the proving capabilities of the *Predicate Logic Prover* [13] of the *Theorema* system. Details on the implementation are given in Section 2.4.

In our approach, for proving the correctness of programs we proceed as follows. First, we formulate the correctness statement. The correctness statement involves the semantics of the program. However, the semantics of the program, being expressed as an implicit definition of a function, it might be contradictory to the object theory. Therefore, the existence and uniqueness of the function implemented by the program has to be proved beforehand. The proof uses a witness which is expressed in terms of the recursion index and of the repetition function, whose existence has to be proved separately. Finally, we prove the correctness statement from the verification conditions.

Summarizing, in order to prove the correctness statement, we need to prove beforehand the soundness of the method, that is:

1. existence of the repetition function,
2. existence of the recursion index,
3. existence and uniqueness of the function implemented by the loop.

Remark 2.13. Note that the meta-functions defined bellow do not apply to programs with nested recursion and for recursive programs containing `while` loops, since this would lead to nested recursion. In this case, the semantics function would occur in the termination condition, fact which we do not allow. In order for our approach to be still applicable, one can eliminate the unwanted occurrences of the function by using new (universally quantified) variables pre-conditioned by the output condition as in Definition 2.8.5.

In the following, let n, m be natural numbers and $^+$ the successor function.

Lemma 2.14. *(Existence of the repetition function) Formula*

$$\underset{h}{\forall} \underset{G}{\exists} \underset{x}{\forall} \left(G[0,x] = x \ \wedge \underset{n\in\mathbb{N}}{\forall}(G[n^+,x] = h[G[n,x]])\right)$$

is a logical consequence of the natural number theory.

Proof. Let x be arbitrary but fixed. First we prove:

$$\underset{h}{\forall}\ \underset{m\in\mathbb{N}}{\forall}\ \underset{H}{\exists}\ \big(H[0]=x \wedge \underset{n<m}{\forall} H[n^+]=h[H[n]]\big)$$

by natural induction on m.
Base Case. We need to prove:

$$\underset{H}{\exists}\left(H[0]=x \wedge \underset{m<0}{\forall} H[n^+]=h[H[n]]\right).$$

The proof is immediate by taking $H[0]=x$.
Induction Step. We assume $\underset{H}{\exists}\left(H[0]=x \wedge \underset{n<m}{\forall} H[n^+]=h[H[n]]\right)$.

We need to prove $\underset{H}{\exists}\left(H[0]=x \wedge \underset{n<m^+}{\forall} H[n^+]=h[H[n]]\right)$.

The proof is immediate for $n<m$. For $m=n$, we take $H[m^+]=h[H[m]]$.
By Skolemization on H one obtains

$$\underset{h}{\forall}\ \underset{\mathcal{H}}{\exists}\ \underset{m\in\mathbb{N}}{\forall}\ \big(\mathcal{H}[m][0]=x\ \wedge \underset{n<m}{\forall}\mathcal{H}[m][n^+]=h[\mathcal{H}[m][n]]\big). \tag{2.7}$$

By (2.7) we have the following

$m\backslash n$	0	1	2	3	...
0	x	x	x	x	...
1	-	$h[x]$	$h[x]$	$h[x]$	...
2	-	-	$h^2[x]$	$h^2[x]$	...
3	-	-	-	$-$	...
...	...	...	...	...	...

The above motivate us to prove

$$\underset{n\in\mathbb{N}}{\forall}\ \underset{m\geq n}{\forall}\ \mathcal{H}[m][n]=\mathcal{H}[n][n]$$

by natural induction on n.
Base Case. We need to prove:

$$\underset{m\geq 0}{\forall}\mathcal{H}[m][0]=\mathcal{H}[0][0] \qquad \text{(by (2.7))}.$$

Induction Step. We assume

$$\underset{m\geq n}{\forall}\mathcal{H}[m][n]=\mathcal{H}[n][n].$$

We need to prove

$$\underset{m\geq n}{\forall}\mathcal{H}[m][n^+]=\mathcal{H}[n^+][n^+].$$

Let m be arbitrary but fixed. We need to prove

$$\underset{m\geq n}{\forall} \mathcal{H}[m][n^+] = \mathcal{H}[n^+][n^+].$$

But

$$\mathcal{H}[m][n^+] \overset{\text{by (2.7)}}{=\!=\!=} h[\mathcal{H}[m][n]] \overset{\text{by Ind. Hypoth.(2.7)}}{=\!=\!=\!=\!=\!=} h[\mathcal{H}[n][n]]$$
$$\overset{\text{by Ind. Hypoth.}}{=\!=\!=\!=} h[\mathcal{H}[n^+][n]] \overset{\text{by (2.7)}}{=\!=\!=} \mathcal{H}[n^+][n^+].$$

By taking $g[n] = \mathcal{H}[n][n]$ one has (since x was arbitrary)

$$\underset{x}{\forall}\ \underset{g}{\exists} \left(g[0] = x \ \wedge \ \underset{n\in\mathbb{N}}{\forall}\ g[n^+] = h[g[n]]\right),$$

which by Skolemization on g gives the desired formula (with notation $G[n, x]$ instead of $G[x][n]$). □

Remark 2.15. We use $h^n[x]$, instead of $G[n][x]$, in our formalism.

Remark 2.16. It is straightforward to show that $h^n[h[x]] = h^{n^+}[x]$.

2.3.1. Correctness of Single Recursive Programs

Single recursive programs are programs with at most one recursive call on each program branch. Such programs have the simplified form (2.8), where Q is a predicate and S, C, and R are functions defined using the constructs present in the program text, possibly using conditionals but no recursion. We assume that f is augmented with the specification $I_f[\alpha]$ and $O_f[\alpha, \beta]$.

$$f[\alpha] := \underline{\texttt{if}} \quad Q[\alpha] \quad \underline{\texttt{then}} \quad \alpha{:=}S[\alpha] \quad \underline{\texttt{else}} \quad \alpha{:=}C[\alpha, f[R[\alpha]]] \tag{2.8}$$

The recursive program (2.8) has the semantics (2.9), the functional verification condition (2.10), and the termination condition (2.11).

$$\underset{\alpha:I_f[\alpha]}{\forall} f[\delta] = \begin{cases} S[\alpha] & \text{if} \quad Q[\alpha] \\ C[\alpha, f[R[\alpha]]] & \text{if} \quad \neg Q[\alpha] \end{cases} \tag{2.9}$$

$$\underset{\alpha:I_f[\alpha],y}{\forall} \bigwedge \begin{cases} Q[\alpha] \Rightarrow O_f[\alpha, S[\alpha]] \\ \neg Q[\alpha] \wedge O_f[R[\alpha], y] \Rightarrow O_f[\alpha, C[\alpha, y]] \end{cases} \tag{2.10}$$

$$\underset{\alpha:I_f[\alpha]}{\forall} \bigwedge \left\{ \begin{array}{l} Q[\alpha] \Rightarrow \pi[\alpha] \\ \neg Q[\alpha] \wedge \pi[R[\alpha]] \Rightarrow \pi[\alpha] \end{array} \right\} \Rightarrow \underset{\alpha:I_f[\alpha]}{\forall} \pi[\alpha] \tag{2.11}$$

The total correctness formula for the program (2.8) is expressed as follows. “Formula $\underset{\alpha:I_f[\alpha]}{\forall} O_f[\alpha, f[\alpha]]$ is a logical consequence of the semantics and verification conditions.”

However, this always holds in the case the semantics is contradictory to the theory, which may happen when the program is recursive. Therefore, one proves first that the existence and uniqueness of an f satisfying the semantics formula is a logical consequence of the verification conditions.

The subsequent properties need the theory of natural numbers, although we do not specify this explicitly.

Lemma 2.17. *(Existence of the recursion index) Formula* $\underset{\alpha:I_f[\alpha]}{\forall}\ \underset{n\in\mathbb{N}}{\exists}\ Q[R^n[\alpha]]$ *is a logical consequence of the termination condition* (2.11) *and the safety verification conditions.*

Proof. The proof uses the induction principle given in (2.11), where $\pi[\alpha]$ is $\underset{n\in\mathbb{N}}{\exists} Q[R^n[\alpha]]$. One needs to use the safety conditions and the property of h^n given above. $\square$

Remark 2.18. One can define now a function (the recursion index of α) $M[\alpha] = \{n \mid Q[R^n[\alpha] \wedge \underset{m\in\mathbb{N}}{\forall}(Q[R^m[\alpha]] \Rightarrow m \geq n)\}$ because the set is nonempty.

Remark 2.19. It is straightforward to show that $M[R[\alpha]]^+ = M[\alpha]$.

Theorem 2.20. *(Existence of the function implemented by the program) Formula* (2.9) *is a logical consequence of the termination condition* (2.11) *and the safety verification conditions.*

Proof. The proof is similar to the one from Lemma 2.14, only that instead of the running argument n we use α with a certain recursion index.

One proves first:

$$\underset{m\in\mathbb{N}}{\forall}\ \underset{F}{\exists}\ \underset{\alpha:I_f[\alpha]}{\forall}(M[\alpha] \leq m) \Rightarrow \big((Q[\alpha] \Rightarrow F[\alpha] = S[\alpha]\big) \wedge \big(\neg Q[\alpha] \Rightarrow F[\alpha] = C[\alpha, F[R[\alpha]]]\big)\big) \tag{2.12}$$

by natural induction on m. By Skolemizing F from (2.12) one obtains:

$$\begin{aligned}&\underset{\mathcal{F}}{\exists}\ \underset{m\in\mathbb{N}}{\forall}\ \underset{\alpha:I_f[\alpha]}{\forall}(M[\alpha] \leq m) \Rightarrow \\ &\qquad \big((Q[\alpha] \Rightarrow \mathcal{F}[m][\alpha] = S[\alpha])\ \wedge (\neg Q[\alpha] \Rightarrow \mathcal{F}[m][\alpha] = C[\alpha, \mathcal{F}[m][R[\alpha]]])\big)\end{aligned}$$

Furthermore one can prove $\underset{\alpha:I_f[\alpha]}{\forall}\ \underset{m\in\mathbb{N}}{\forall}(m \geq M[\alpha])\ \Rightarrow (\mathcal{F}[m][\alpha] = \mathcal{F}[M[\alpha]][\alpha])$ by the induction given in the formula (2.11) (taking as $\pi[\alpha]$ the formula above without the quantifier for α).

Finally one takes $f[\alpha] = \mathcal{F}[M[\alpha]][\alpha]$. $\square$

Remark 2.21. Uniqueness of f is straightforward: take f_1, f_2 satisfying (2.9) and use (2.11) with $\pi[\alpha]$ as $f_1[\alpha] = f_2[\alpha]$.

Theorem 2.22. *(Total correctness) Formula* $\underset{\alpha:I_f[\alpha]}{\forall} O_f[\alpha, f[\alpha]]$ *is a logical consequence of the program semantics and the verification conditions.*

Proof. The proof is straightforward by taking in (2.11) $\pi[\alpha]$ as $O_f[\alpha, f[\alpha]]$. This is because the left-hand side of the (2.11) becomes identical to the functional verification condition 2.10. □

2.3.2. Correctness of Simple Loops

In this section, we prove the correctness of loops which can be brought into the following form:

$$\underline{\texttt{while}} \quad \phi[\delta] \quad \underline{\texttt{do}} \quad \delta{:=}R[\delta], \tag{2.13}$$

annotated with the loop invariant $\iota[\delta]$, where $\phi[\delta]$, and $R[\delta]$ are the loop condition and the function representing the update of the critical variable δ performed in the loop body, respectively. They are defined using the constructs present in the program text, possibly using conditionals but no recursion. For example, in Algorithm 2, $\phi[i]$ is $i < n$ and $R[i]$ is $i + 1$.

<u>While</u> loop (2.13) has the semantics (2.14), partial correctness (safety) condition (2.15) and termination condition (2.16).

$$\underset{\delta:\iota[\delta]}{\forall} f[\delta] = \begin{cases} \delta & \text{if} \quad \neg\phi[\delta] \\ f[R[\delta]] & \text{if} \quad \phi[\delta] \end{cases} \tag{2.14}$$

$$\underset{\delta:\iota[\delta]}{\forall} \iota[R[\delta]] \tag{2.15}$$

$$\underset{\delta:\iota[\delta]}{\forall} \wedge \left\{ \begin{array}{c} \neg\phi[\delta] \Rightarrow \pi[\delta] \\ \phi[\delta] \wedge \pi[R[\delta]] \Rightarrow \pi[\delta] \end{array} \right\} \Rightarrow \underset{\delta:\iota[\delta]}{\forall} \pi[\delta] \tag{2.16}$$

The total correctness statement of simple <u>while</u> loops "The loop invariant is always preserved." is expressed formally as:

$$\underset{\delta:\iota[\delta]}{\forall} \iota[f[\delta]]. \tag{2.17}$$

We express the soundness of the verification method for loops of type (2.13) as follows: "Formula (2.17) is a logical consequence of the semantics (2.14) and of the termination condition (2.16)."

Note that a function like in (2.14) always exists but does not necessary terminate. However, we still prove explicitly its existence (and uniqueness) based on a witness term. The fact that the witness has a closed-form solution is important for the simplicity of the proofs.

The subsequent properties need the theory of natural numbers, although we do not specify it explicitly.

Lemma 2.23. *(Existence of the recursion index) Formula*

$$\underset{\delta:\iota[\delta]}{\forall}\ \underset{n}{\exists}\Big(\neg\phi[R^n[\delta]] \wedge \underset{m}{\forall}\ \big(\neg\phi[R^m[\delta]] \Rightarrow m \geq n\big)\Big)$$

is a logical consequence of the termination condition (2.16)*.*

Proof sketch. The automated proof uses a built-in natural induction principle. Additionally, the following assumptions are used:

$$\underset{x}{\forall}\ R^0[x] := x \tag{2.18a}$$

$$\underset{x,n}{\forall}\ R^n[R[x]] := R^{n^+}[x] \tag{2.18b}$$

$$\underset{n}{\forall}\ n \geq 0 \tag{2.18c}$$

$$\underset{n\neq 0}{\forall}\ (n^-)^+ := n \tag{2.18d}$$

$$\underset{m,n}{\forall}\ m \geq n \Rightarrow m^+ \geq n^+ \tag{2.18e}$$

Note that there are two types of premises used in the proof:

1. properties of the repetition function R: (2.18a), (2.18b),
2. properties of the natural number theory: (2.18c), (2.18d). (2.18e), (2.16).

We are asking the question: can we trust them? It is obvious that (2.18a) and (2.18b) satisfy the properties of the function h from Lemma 2.14. Using a model of $\mathbb{N}$ involving the constant 0, the functions S (successor function) and + (plus function) and the axioms (2.19a) - (2.19e), the definitions (2.18c), (2.18d), (2.18e) and (2.16) can be derived.

$$S(x) \neq 0 \tag{2.19a}$$

$$(S(x) = S(y)) \Rightarrow (x = y) \tag{2.19b}$$

$$\underset{P}{\forall}\ \Big(P(0) \wedge \big(\underset{k}{\forall}\ (P(k) \Rightarrow P(S(k)))\big) \Rightarrow \underset{n}{\forall}\ P(n)\Big) \tag{2.19c}$$

$$x + 0 = x \tag{2.19d}$$

$$x + S(y) = S(x + y) \tag{2.19e}$$

However, these definitions do not characterize $\mathbb{N}$ completely, in particular one can not define the $\leq$ relation in the usual sense, i.e. $3 \leq 4$. But using these axioms the proof of Lemma 2.23 succeeds. Hence, it succeeds for any relation satisfying (2.18c), (2.18d), (2.18e), and (2.16), in particular, for the minimal relation needed, which is the order relation on $\mathbb{N}$.

In Appendix A.1, we present the *Theorema* generated proof of Lemma 2.23.

Remark 2.24. From Lemma 2.23, one can see immediately that n is unique, thus, by Skolemization, one obtains the function $M[\delta]$ called the recursion index of δ, that is: $M[\delta] := \{n \mid (\neg\phi[R^n[\delta]] \wedge \underset{m\in\mathbb{N}}{\forall} (\neg\phi[R^m[\delta]] \Rightarrow m \geq n)\}$.

Lemma 2.25. *(Existence and uniqueness of the function implemented by the loop) The existence and uniqueness of an f satisfying formula* (2.14) *is a logical consequence of the termination condition* (2.16) *and of the safety verification condition* (2.15).

Proof sketch. For proving the existence, one takes $\underset{\delta:\iota[\delta]}{\forall} f[\delta] := R^{M[\delta]}[\delta]$ as witness for the loop semantics and derives the expression of f on each execution program branch as required by (2.14) The proof requires also the use of (2.18a), (2.18b) and:

$$\underset{\delta:\iota[\delta]}{\forall} (\neg\phi[\delta] \Rightarrow M[\delta] := 0) \tag{2.20a}$$

$$\underset{\delta:\iota[\delta]}{\forall} (M[R[\delta]]^+ := M[\delta]) \tag{2.20b}$$

For proving the uniqueness, one takes two different semantics functions, e.g. f and g, of the form (2.14) and shows that they are the same. The key in this proof is the instantiation in the termination condition of $\pi[\delta]$ with $f[\delta] = g[\delta]$.

We present the *Theorema* generated proof of Lemma 2.25: the existence in Appendix A.2 and the uniqueness in Appendix A.3.

Remark 2.26. Note that a total function f as in (2.14) always exists, but it is not necessarily unique. Its uniqueness comes from the termination condition.

Theorem 2.27. *(Correctness of simple loops) Formula* (2.17) *is a logical consequence of the semantics formula* (2.14) *and of the termination condition* (2.16).

Proof sketch. The proof is straightforward by taking in (2.16) $\pi[\delta]$ as $\iota[f[\delta]]$. This is because the left-hand side of the (2.16) becomes identical to the functional conditions generated for partial correctness.

The *Theorema* proof of Theorem 2.27 is listed in Appendix A.4.

Remark 2.28. Theorem (2.27) can be proved also by using the semantics witness from Theorem 2.25. In the respective proof, one needs information about the loop semantics on different execution program branches as given by (2.14).

2.3.3. Correctness of Abruptly Terminating Loops

There are basically two methods of proving the correctness of an abruptly terminating loop:

1. prove its correctness directly;
2. transform it into an equivalent simple one and prove the total correctness of the transformed version.

The drawback in the first case is that the invariant might become difficult to express and too lengthy for loops with many ramifications and abrupt statements. In the second case, the difficulty might arise at program transformation, but the gain is that the invariants are simpler and the correctness of the initial loop resumes to the correctness of a loop-free construct due to the fact that the correctness of simple loops was already proved (Section 2.3.2).

We chose to prove correctness by the second method.

Non-nested Abruptly Terminating Loops. Case `break`

Any <u>`while`</u> loop abruptly terminating via <u>`break`</u> can be expressed as in Example 2.29 even if it contains other loops with <u>`break`</u>; <u>`break`</u> from a inner loop can be eliminated and the inner loop can be expressed as a function call.

Example 2.29.

<u>`while`</u> $\phi[\delta]$ <u>`do`</u>
  <u>`if`</u> $\psi[\delta]$ <u>`then`</u>
    $\delta := S[\delta]$;
    <u>`break`</u>
  <u>`else`</u>
    $\delta := R[\delta]$

Example 2.30.

<u>`while`</u> $\phi[\delta] \wedge \neg\psi[\delta]$ <u>`do`</u>
    $\delta := R[\delta]$;
<u>`if`</u> $\phi[\delta] \wedge \psi[\delta]$ <u>`then`</u>
    $\delta := S[\delta]$

For instance, Example 2.29 is transformed into Example 2.30.

Each loop is annotated with an invariant. Note that, in general, the invariants of Examples 2.29 and 2.30 are not the same, namely the invariant of Example 2.29 is stronger. However, we use this invariant for both loops (and we refer to it as $\iota[\delta]$) because it implies also the invariant of the loop in Example 2.30. The same holds for Examples 2.29 and 2.30.

Like for simple loops, the correctness of abruptly terminating <u>`while`</u> loops via <u>`break`</u> resumes to proving the soundness of the method. In this case, proving soundness reduces to show the equivalence of semantics functions of Examples 2.29 and 2.30 (Lemma 2.31). Let (2.21) be the semantics of Example 2.29 and (2.22) a witness satisfying it.

$$\underset{\delta:\iota[\delta]}{\forall} f[\delta] = \begin{cases} \delta & \text{if } \neg\phi[\delta] \\ S[\delta] & \text{if } \ \ \phi[\delta] \wedge \psi[\delta] \\ f[R[\delta]] & \text{if } \ \ \phi[\delta] \wedge \neg\psi[\delta] \end{cases} \tag{2.21}$$

$$\underset{\delta:\iota[\delta]}{\forall} f[\delta] := \begin{cases} R^{M[\delta]}[\delta] & \text{if } \neg(\phi[R^{M[\delta]}[\delta]] \wedge \psi[R^{M[\delta]}[\delta]]) \\ S[R^{M[\delta]}[\delta]] & \text{if } \ \ \phi[R^{M[\delta]}[\delta]] \wedge \psi[R^{M[\delta]}[\delta]] \end{cases} \tag{2.22}$$

where

$$M[\delta] := \Big\{ n | \neg\left(\phi[R^n[\delta]] \wedge \neg\psi[R^n[\delta]]\right) \wedge \big(\underset{m\in\mathbb{N}}{\forall} \neg\left(\phi[R^m[\delta]] \wedge \neg\psi[R^m[\delta]]\right) \Rightarrow m \geq n \big) \Big\}$$

is the recursion index of the loop. Further, let (2.23) and (2.24) be the semantics of the simple loop and, respectively, of the conditional obtained of Example 2.30.

$$\underset{\delta:\iota[\delta]}{\forall} f'[\delta] = \begin{cases} \delta & \text{if } \neg(\phi[\delta] \wedge \neg\psi[\delta]) \\ f'[R[\delta]] & \text{if } \quad \phi[\delta] \wedge \neg\psi[\delta] \end{cases} \tag{2.23}$$

$$\underset{\delta:\iota[\delta]}{\forall} g'[\delta] = \begin{cases} \delta & \text{if } \neg\phi[\delta] \\ S[\delta] & \text{if } \quad \phi[\delta] \wedge \psi[\delta] \end{cases} \tag{2.24}$$

Let (2.25) and (2.26) be witnesses satisfying (2.23), respectively, (2.24).

$$\underset{\delta:\iota[\delta]}{\forall} f'[\delta] := R^{M[\delta]}[\delta] \tag{2.25}$$

$$\underset{\delta:\iota[\delta]}{\forall} g'[\delta] := \begin{cases} \delta & \text{if } \neg(\phi[\delta] \wedge \psi[\delta]) \\ S[\delta] & \text{if } \quad \phi[\delta] \wedge \psi[\delta] \end{cases} \tag{2.26}$$

The semantics witness of Example 2.30 is $F'[\delta] = g'[f'[\delta]]$ and is obtained by composing the semantics witnesses (2.25) and (2.26). We have

$$\underset{\delta:\iota[\delta]}{\forall} F'[\delta] := \begin{cases} R^{M[\delta]}[\delta] & \text{if } \neg(\phi[R^{M[\delta]}[\delta]] \wedge \psi[R^{M[\delta]}[\delta]]) \\ S[R^{M[\delta]}[\delta]] & \text{if } \quad \phi[R^{M[\delta]}[\delta]] \wedge \psi[R^{M[\delta]}[\delta]]) \end{cases} \tag{2.27}$$

Lemma 2.31. *Examples 2.29 and 2.30 implement the same semantics function.*

Proof sketch. The proof is immediate by observing that (2.22) and (2.27) are the same.

Non-nested Abruptly Terminating Loops. Case `return`

Any <u>`while`</u> loop abruptly terminating via <u>`return`</u> can be expressed as in Example 2.32 even if it contains other loops with <u>`break`</u> and <u>`return`</u>; both <u>`break`</u> and <u>`return`</u> from a inner loop can be eliminated and the inner loop can be expressed as function call.

Example 2.32.

<u>`while`</u> $\phi[\delta]$ <u>`do`</u>
  <u>`if`</u> $\psi[\delta]$ <u>`then`</u>
    $\delta := S[\delta]$;
    <u>`return`</u> *[δ]*
  <u>`else`</u>
    $\delta := R[\delta]$

Example 2.33.

<u>`while`</u> $\phi[\delta] \wedge \neg\psi[\delta]$ <u>`do`</u>
    $\delta := R[\delta]$;
<u>`if`</u> $\phi[\delta] \wedge \psi[\delta]$ <u>`then`</u>
    $\delta := S[\delta]$
    <u>`return`</u> *[δ]*

For instance, Example 2.32 is transformed into Example 2.33.

The correctness of loops abruptly terminating via <u>`return`</u> can be proved following the principles of loop abruptly terminating via <u>`break`</u>, with the remark that the <u>`return`</u>

statement causes execution to exit the program. Hence, additionally to proving the equivalence of the semantics functions of Examples 2.32 and 2.33, one has to prove that the output condition of the program holds upon the execution of the <u>`return`</u> (see Appendix B.1)

Nested Abruptly Terminating Loops

Our approach can be extended to arbitrarily nested, abruptly terminating `while` loops. The proofs are similar to those with non-nestedness, the effort is to transform the initial loops into simple loops. A naive algorithm for such a translation is:

1. analyze the program top-down detecting the innermost loop with abrupt termination,
2. transform it into a normal terminating one; the abrupt statements are eliminated in the order they appear: <u>`break`</u> is eliminated from the currently analyzed loop, from the all wrapper loops and from the program itself, <u>`return`</u> is eliminated only from the currently analyzed loop,
3. repeat 1. and 2. until there are no loops with abrupt termination,
4. the program text which does not need transformations is copied correspondingly.

We apply our algorithm to Algorithm 3. The translated version is as follows.

$$
\begin{array}{l}
i := 0;\ j := 0;\\
\underline{\texttt{while}}\ \Big(i < m \wedge \neg\big(j < n \wedge (e = a[i][j])\big)\Big)\ \underline{\texttt{do}}\\
\quad j := 0;\\
\quad \underline{\texttt{while}}\ \big(j < n \wedge (e \neq a[i][j])\big)\ \underline{\texttt{do}}\\
\quad\quad j := j + 1;\\
\quad i := i + 1;\\
\underline{\texttt{if}}\ ((i < m \wedge j < n \wedge (e = a[i][j]))\ \underline{\texttt{then}}\ \underline{\texttt{return}}[\langle i, j\rangle];\\
\underline{\texttt{return}}\ [-1];
\end{array}
$$

The abrupt termination via <u>`return`</u> was transferred to the main program. The correctness of the simple loops is proved as follows:

1. prove the correctness of the inner loop,
2. prove the correctness of the wrapper loop by considering the inner loop as a black-box characterized by the loop invariant; the loop invariant is used in the proof of correctness of the wrapper loop.

2.4. Implementation

The formalization, implementation, and automated proof of soundness of our verification method are performed in the *Theorema* system. The system was built with the goal of providing one logical and software system frame for the entire process of mathematical exploration process.

Theorema is a computer aided mathematical software which is being developed at Research Institute for Symbolic Computation (RISC) in Hagenberg, Austria. The system offers support for *computing, proving* and *solving* mathematical expressions using specified knowledge bases by applying several simplifiers, solvers and provers in natural style, which imitate the heuristics used by human provers. Composing, structuring and manipulating mathematical texts is also possible in the system using labeling (Definition, Theorem, Proposition). For our research (program verification), it is very important that the *Theorema* system provides a very expressive way to define *algorithms*: they are written in the language of predicate logic with equality as rewrite rule. *Theorema* provides elegant proofs (because of natural style inferences used) in the verification process of programs. Moreover, being built on top of the computer algebra system *Mathematica*, it has access to many computing and solving algorithms.

2.4.1. The Theorema System

Theorema system aims at providing a uniform framework for computing, solving, and proving. It is built on top of the *Mathematica* computer algebra system [92], thus it uses many features of the language. The features important for our research are enumerated as follows.

1. The core part of *Mathematica* language is higher-order equational logic. Hence, *Mathematica* can be considered as the "logic-internal" programming language of *Theorema*.
2. The rule-based programming style of *Mathematica* is used for the implementation of provers (in particular *Predicate Logic Prover*) and the program analyzers (in particular *FwdVCG*), which are basically a list of rewrite rules.
3. *Mathematica* provides the "notebook facility". Notebooks are utilized in the phases of problem specification, programming, and proving.

Theorema provides support in all cycles of development of mathematical activity through *language layers*.

2.4.2. Theorema Language Layers

The following language layers are available in *Theorema*: *writing mathematical statements*, *formalization of mathematics*, and *mathematical activities*.

Writing Mathematical Statements in *Theorema*. *Theorema* expression language is a version of *higher-order predicate logic without extensionality.* The ingredients of the language are: constants, variables, terms, predicates, quantifiers.

Formalization of Mathematics. Besides writing mathematical statements, *Theorema* allows the built-up of *theories* by formulating new concepts through *definitions*, by stating new properties through *theorems, lemmas, propositions.*

Supported Mathematical Activities. After building-up the knowledge base, *Theorema* allows: *i*) *proving* the theorems that have been stated; *ii*) *computing* examples using specified knowledge; *iii*) *solving* problems.

For the exemplification of *Theorema* language layers, we prove Theorem 2.27 in the system. The necessary notions are introduced in the system as follows.

`Definition`["Termination",

$$\left[\left(\underset{\substack{\delta \\ \iota[\delta]}}{\forall}\left((\neg\phi[\delta] \Rightarrow \pi[\delta]) \wedge (\phi[\delta] \wedge \pi[R[\delta]] \Rightarrow \pi[\delta])\right)\right) \Rightarrow \left(\underset{\substack{\delta \\ \iota[\delta]}}{\forall} \pi[\delta]\right) \quad \texttt{""}\right]$$

`Definition`["Semantics",

$$\left.\underset{\substack{\delta \\ \iota[\delta]}}{\forall}\left((\neg\phi[\delta] \Rightarrow (f[\delta]{:=}\delta)) \wedge (\phi[\delta] \Rightarrow (f[\delta]{:=}f[R[\delta]]))\right) \quad \texttt{""}\right]$$

`Assumption`["Instantiation of π",

$$\left.\underset{\delta}{\forall}(\pi[\delta] :\Leftrightarrow \iota[f[\delta]]) \quad \texttt{""}\right]$$

2.4.3. Predicate Logic Prover. Extension

We are interested in proving, activity which is realized internally in the system by a prover. We present the mechanism of the *Predicate Logic Prover*, the prover that we extended for program verification purposes.

The *Predicate Logic Prover* [13] deals with *proof situations* consisting of (higher-order) predicate logic formulas. A *proof situation* consists of a *goal* and a *knowledge base.* For reference purpose, the initial knowledge base, the goal, as well as all the newly generated formulas are *labeled.* New formulas are generated using *inference rules.* The decision which inference rule is applied is taken by inspecting the outermost symbols of the goal and of the knowledge base. Therefore, the prover is basically a *sequent calculus.* However, since the goal of the prover is not its completeness, rather *natural style proofs*, the sequent calculus implemented by the prover contains more inference rules than the ones presented in the logic books, the later ones focusing on a minimal set.

Proving with the *Predicate Logic Prover* is invoked using the command:

> **Prove**[Theorem[T], [by → PredicateProver,] using → KB,
> [ProverOptions → {BackChaining → True},] TransformerOptions → {steps →
> Useful}, SearchDepth → n],

meaning that the *Theorem T* is tried to be proved using the *knowledge base KB* (specified by using the built-in constructs like Definition, Assumption, etc.), using *PredicateProver*. If PredicateProver was set as the default prover then the option "by → PredicateProver" can be omitted. If the goal to be proved is exactly the right hand side of an implication in the KB, the strategy is to try to prove first the left hand side of the respective implication. This is possible in the *Theorema* system by enabling the option *BackChaining*. The main reason for trying this strategy at the very beginning of the proof is that the proof is delivered in natural style. Further, the option *TransformerOptions→{steps→Useful}* is used for esthetic reasons: only the facts and inferences necessary for the final proof are displayed. *SearchDepth* option specifies the maximal search depth n in in the proof tree.

For example, we want to prove the total correctness of simple loops, that is:

`Theorem`["Total Correctness",
$$\underset{\iota[\delta]}{\underset{\delta}{\forall}}\ \iota[f[\delta]] \quad ""\ ]$$

using the knowledge base Definition["Termination"], Definition["Semantics"], and Assumption["Instantiation of π"]. This can be done in the system by issuing the command:

```
Prove[Theorem["Total Correctness"],
  using → {
      Definition["Termination"],
      Definition["Semantics"],
      Assumption[ "Instantiation of π"]},
  ProverOptions → {BackChaining → True},
  TransformerOptions → {steps → Useful},
  SearchDepth → 70];
```

The prover is implemented as a set of inference rules, typically expressed as rewrite rules transforming the proof situation into one or more new proof situations. Proofs are internally represented by *proof objects*, containing the complete history of inference rule applications. As the prover continues, the proof object is expanded from the initial proof situation to a tree representing the full proof (in case of success). At certain points, a proof situation is split into two, and the prover continues, either by trying to prove both newly created proof situations (e.g. when proving a conjunction), or one of them (e.g. when proving a disjunction).

We describe briefly how the proof is stored internally. From the *initial proof situation*, a *proof object* is constructed containing the proof situation. In each proof step this proof object is extended until it contains the *proof*, that is all information on intermediary proof situations and inference rules applied at each step. The proof object is not accessible to the user, however it contains all the information necessary to produce a proof in natural language. Natural language facility is available in *Theorema*, however the proof object can be used also by other systems once a suitable translation is available. A proof object has one of the forms presented below.

1. `PND[formula, knowledge-base]`, where `PND` is the internal name of the *Predicate Logic Prover* (`PND` = "Proof by Natural Deduction"), is an evaluated proof object containing only the proof situation: `formula` is the labeled formula to be proved, `knowledge-base` is the knowledge base consisting of axioms, definitions, etc. Because the prover is a set of rewrite rules of the form `PND[proof-situation] := proof-object`, *Mathematica* will try apply one of the inference rules to the unevaluated proof object. This is also the working mechanism of the prover.
2. ⟨`proof-rule-info, list-of-proof-objects, proof-result`⟩ is a (partially) evaluated proof object, where
   - `proof-rule-info` is the information which prover is applied in the current proof situation, namely *name of the prover*, the (labels of the) formulas involved in this proof situation, and new formulas formed in this proof situation;
   - `list-of-proof-objects` describes the subproofs generated by applying the prover in the current proof situation. In case the list is empty, then we deal with a terminal proof object meaning that the proof was completed on a branch
   - `proof-result` contains information whether the proof was successful or not. If it is missing then the proof object still contains unevaluated subobjects. However, it is always present in a terminal proof.

Once the proof object is constructed, the *Theorema* system transforms it into a natural language proof which is displayed in a separate notebook file. The *Theorema* proof of Theorem 2.27 using *PredicateProver*, the knowledge base and options specified previously is listed in Appendix A.4.

For the *Theorema* proof of Theorem 2.27, as well as for all the other automated proofs of the soundness of the verification method, the *Predicate Logic Prover* (implemented in *Mathematica* 5.2) had to be enhanced with new inference rules. Moreover, a significant effort was required to develop a minimal set of inference rules for the success of the proof. Since the pattern matching mechanism of *Mathematica* applies the inference rules in the order they appear, we had take into account this fact when implementing the proving strategy.

In the following, we briefly describe the proving strategy. First we check if the goal is already between the assumptions (terminal proof situation). If not, we derive new assumptions using the existing knowledge base and inference rules, both for propositional and first-order formulas. Afterwards, we try to simplify the goal using, first, propositional inferences, and, second, first-order inferences. After that, the goal is decomposed, if that is the case. The inference rules are repeatedly applied until the goal has been proved.

The prover uses well-known inference rules like e.g. deduction rule, decomposition of conjunction in the goal and in the assumptions, modus ponens.

In the following, we list and exemplify the most important inference rules. When a new rule was added to the prover or the existing ones were modified, we specify that fact.

1. *Back chaining on the goal.* If the prover has switched on the option BackChaining, then we try to prove the left hand side of the implication. This might not succeed, but if it does, the proofs look nicer. Example:

   Prove:

   (Theorem (Total Correctness))

   $$\underset{\delta}{\forall}(\iota[\delta] \Rightarrow \iota[f[\delta]]),$$

   under the assumptions:

   (Definition (Termination))

   $$\underset{\delta}{\forall}(\iota[\delta] \Rightarrow (\neg\phi[\delta] \Rightarrow \pi[\delta]) \wedge (\phi[\delta] \wedge \pi[R[\delta]] \Rightarrow \pi[\delta])) \Rightarrow \underset{\delta}{\forall}(\iota[\delta] \Rightarrow \pi[\delta]),$$

   ...

   (Assumption (Instantiation of π))

   $$\underset{\delta}{\forall}(\pi[\delta] :\Longleftrightarrow \quad \iota[f[\delta]]).$$

   From (Definition (Termination)), by (Assumption (Instantiation of π)), we obtain:

   (1) $\underset{\delta}{\forall}(\iota[\delta] \Rightarrow (\neg\phi[\delta] \Rightarrow \iota[f[\delta]]) \wedge (\phi[\delta] \wedge \iota[f[R[\delta]]] \Rightarrow \iota[f[\delta]])) \Rightarrow \underset{\delta}{\forall}(\iota[\delta] \Rightarrow \iota[f[\delta]])$.

   For proving (Theorem (Total Correctness)), by (1), it suffices to prove

   (3) $\underset{\delta}{\forall}(\iota[\delta] \Rightarrow (\neg\phi[\delta] \Rightarrow \iota[f[\delta]]) \wedge (\phi[\delta] \wedge \iota[f[R[\delta]]] \Rightarrow \iota[f[\delta]]))$.

   ...

2. *Elimination of universal quantification in the goal.* Universally quantified variables in the goal become Skolem constants (arbitrary, but fixed). Example:

   ...

   (3) $\underset{\delta}{\forall}(\iota[\delta] \Rightarrow (\neg\phi[\delta] \Rightarrow \iota[f[\delta]]) \wedge (\phi[\delta] \wedge \iota[f[R[\delta]]] \Rightarrow \iota[f[\delta]]))$.

   For proving (3) we take all variables arbitrary but fixed and prove:

   (4) $\iota\,[\delta_0] \Rightarrow (\neg\phi\,[\delta_0] \Rightarrow \iota\,[f\,[\delta_0]]) \wedge (\phi\,[\delta_0] \wedge \iota\,[f\,[R\,[\delta_0]]] \Rightarrow \iota\,[f\,[\delta_0]])$.

   ...

3. *Built-in natural induction principle.* This inference rule was especially added for proving the existence of the recursion index. Example:
...
(7) $\left(\neg\phi\,[\delta_0] \Rightarrow \underset{n}{\exists}\left(\neg\phi\,[R^n\,[\delta_0]] \wedge \underset{m}{\forall}\,(\neg\phi\,[R^m\,[\delta_0]] \Rightarrow m \geq n)\right)\right) \wedge$
$\left(\quad \phi\,[\delta_0] \wedge \underset{n}{\exists}\left(\neg\phi\left[R^{n^+}\,[\delta_0]\right] \wedge \underset{m}{\forall}\left(\neg\phi\left[R^{m^+}\,[\delta_0]\right] \Rightarrow m \geq n\right)\right) \Rightarrow\right.$
$\left.\underset{n}{\exists}\left(\neg\phi\,[R^n\,[\delta_0]] \wedge \underset{m}{\forall}\,(\neg\phi\,[R^m\,[\delta_0]] \Rightarrow m \geq n)\right)\right).$
To prove (7) one has to prove
(8) $\neg\phi\,[\delta_0] \Rightarrow \neg\phi\left[R^0\,[\delta_0]\right] \wedge \underset{m}{\forall}\,(\neg\phi\,[R^m\,[\delta_0]] \Rightarrow m \geq 0)$ and assumes
(9) $\phi\,[\delta_0] \wedge \left(\neg\phi\left[R^{{n_0}^+}\,[\delta_0]\right] \wedge \underset{m}{\forall}\left(\neg\phi\left[R^{m^+}\,[\delta_0]\right] \Rightarrow m \geq n_0\right)\right)$ and proves
(10) $\neg\phi\left[R^{{n_0}^+}\,[\delta_0]\right] \wedge \underset{m}{\forall}\,(\neg\phi\,[R^m\,[\delta_0]] \Rightarrow m \geq {n_0}^+)$.
Hence, when the goal has the form: $\left(\neg f \Longrightarrow \underset{n}{\exists} F[n]\right) \wedge \left(f \wedge \underset{n}{\exists} F[n^+] \Longrightarrow \underset{n}{\exists} F[n]\right)$, one proves $\neg f \Longrightarrow F[0]$ (base case) and, assumes $f \wedge \underset{n}{\exists} F[n^+]$ and proves $\underset{n}{\exists} F[n]$.
4. *Equal by definition in the assumption/goal (among predicates).* An universally quantified formula of the form $u[x] :\Longleftrightarrow\ v[x]$ can be used to rewrite an assumption/a goal of the form $u[t]$. Example: ...
under the assumptions:
(Definition (Termination))
$$\underset{\delta}{\forall}(\iota[\delta] \Rightarrow (\neg\phi[\delta] \Rightarrow \pi[\delta]) \wedge (\phi[\delta] \wedge \pi[R[\delta]] \Rightarrow \pi[\delta])) \Rightarrow \underset{\delta}{\forall}(\iota[\delta] \Rightarrow \pi[\delta]),$$
...
(Assumption (Instantiation of π))
$$\underset{\delta}{\forall}(\pi[\delta] :\Longleftrightarrow\ \iota[f[\delta]]).$$
From (Definition (Termination)), by (Assumption (Instantiation of π)), we obtain:
(1) $\underset{\delta}{\forall}(\iota[\delta] \Rightarrow (\neg\phi[\delta] \Rightarrow \iota[f[\delta]]) \wedge (\phi[\delta] \wedge \iota[f[R[\delta]]] \Rightarrow \iota[f[\delta]])) \Rightarrow \underset{\delta}{\forall}(\iota[\delta] \Rightarrow \iota[f[\delta]])$.
...
5. *Equal by definition in the assumption/goal (among functions).* An universally quantified formula of the form $u[x] := v[x]$ can be used to rewrite an assumption/a goal of the form $u[t]$. Example:
(11) $M\,[\delta_0] := 0$.
Using (7), the goal (5) is transformed into:
(12) $R^{M[\delta_0]}\,[\delta_0] = \delta_0$.
Using (11), the goal (12) is transformed into:
(13) $R^0\,[\delta_0] = \delta_0$.

2.4.4. Adding a Symbolic Execution Feature to the Theorema System

Static program analysis using *symbolic execution* tries to answer the question: *Given a program together with a specification, does the program fulfills the specification?* To answer this question, one usually performs the following tasks:

- annotates the loops with suitable *invariants* (if that is the case)
- the program, the specification, and the invariants are fed into a *verification system*
- which generates *a conjecture*
- which is handled by a *prover*
- and one obtains an *answer* (Yes/No) whether the program fulfills/does not fulfill its specification, and/or a *proof (attempt)* of the conjecture.

The program is expressed in a *programming language*, the specification, invariants, conjecture in a *logic language*, the answer and/or proof (attempt) in the above *logic language* and some *proof language.*

Theorema has its own *language*, hence the specification, invariants, conjectures can be expressed in the *logic language of Theorema*: higher-order predicate logic, including two-dimensional notation. The logic language of *Theorema* together with the constructs which allow creating and structuring the mathematical knowledge form a very accessible *formal language.*

We implemented in *Theorema* an imperative recursive language consisting in the imperative structures introduced in Section 2.2, namely assignments (including recursive call), conditionals, `while` loops with abrupt termination (`break`, `return`), *sequential composition of commands.* Similar to [52], we consider in our approach that a program has a specification part and an implementation part (body). Specification, invariants and conjectures are expressed in the logic language of *Theorema.* The syntax checker and the verification conditions generator working by symbolic execution are integrated in the *Theorema* system.

We mention that *Theorema* already has implemented a simple imperative programming environment with an interpreter and a verifier based on Hoare logic and the weakest precondition strategy [52, 53].

Interface Constructs of the Programming Language

The *interface constructs* of the programming language allow writing programs together with a specification and analyzing them, namely checking the syntactic correctness and generating verification conditions for partial correctness and termination.

1. *Program specification* is introduced by the following command:

   `Specification[label, interface, precondition, postcondition]`,

   where `label` is a name for the specification, `interface` consists of the program name and input parameters, `precondition` and `postcondition` are introduced by, respectively, `Pre` and `Post`. This feature was already implemented in the system, we just used it.

For example, the specification of Algorithm 1 is as follows.

$$\begin{array}{l} \texttt{Specification}[\text{"GCD"}, \text{GCD}[\downarrow a, \downarrow b], \\ \quad \texttt{Pre} \to IsInteger[a] \wedge IsInteger[b] \wedge a \geq 0 \wedge b \geq 0, \\ \quad \texttt{Post} \to \underset{\substack{k \\ IsInteger[k]}}{\exists} \; a = k \cdot out] \end{array}$$

In the above, $\downarrow a$ means that a is an input variable. *out* is the value returned by the program.

2. The *program* consists of program code and the interface definition, and is introduced by the following command:

   `Program[label, interface, code, specification]`,

   consisting in a a name of the program, an interface, the actual code and a specification. `Program` construct was already implemented in the system, we just used it. The program code can be built up using the commands introduced in Section 2.2. The specification must be defined beforehand.

   For example, Algorithm 1 is written in our programming language as follows

$$\begin{array}{l} \texttt{Program}[\text{"GCD"}, \text{GCD}[\downarrow a, \downarrow b], \\ \;\texttt{Module}[, \\ \;\;\texttt{If}[a = 0, \texttt{Return}[b], \\ \;\;\;\texttt{If}[b \neq 0, \\ \;\;\;\;\texttt{If}[a > b, \text{a := GCD[a - b, b]}, \\ \qquad\qquad \text{a := GCD[a, b - a]}]; \\ \;\;\texttt{Return}[a]]]], \\ \;\texttt{Specification} \to \texttt{Specification}[\text{GCD}]] \end{array}$$

   Note that we are using := for variable assignment and = for logical equality. The `Program` construct transforms the program into a list of statements. Omitting the *IsInteger* predicate for simplicity, for Algorithm 1 we have

   •*prog* [...,`Module`[{}, ™`CompoundExpression`[™`If`[$a = 0$, ™Return[b], ™If[$b \neq 0$, ™If[$a > b$, $a :=$ GCD[$a - b, b$], $a :=$ GCD[$a, b - a$]]]]], ™Return[a]]].

   It is important to notice the *quoting* mechanism (marked by the symbol ™) introduced by `Program`, as well as `Specification` and `FwdVCG`, which avoids the evaluation of the expressions in *Mathematica* and allows further reasoning about and manipulation of syntactic structures. • denotes an internal *Theorema* data structure.

3. The *program analyzer* (`FwdVCG` – implemented in *Mathematica* 5.2) takes a `Specification` and a `Program`, checks the syntactical correctness of the program, and generates verification conditions according to the inductive definitions introduced in Sections 2.2.1, 2.2.2, and 2.2.3. Internally, `FwdVCG` operates on •*spec*

and •*prog* *Theorema* data structures and is implemented in *Theorema*.
The analyzer is called by issuing the following command:

```
FwdVCG[Program[label], Specification[label]]
```

For example, Algorithm 1 is analyzed as follows.

```
FwdVCG[Program["GCD"], Specification["GCD"]]
```

The outcome of this command are a list of verification conditions ensuring the partial correctness and termination of the program.
The partial correctness conditions generated are as follows (For simplicity we omitted the integer type of the variables).

$$a \geq 0 \wedge b \geq 0 \wedge a = 0 \implies \underset{k}{\exists}\, a = k \cdot b$$
$$a \geq 0 \wedge b \geq 0 \wedge a \neq 0 \wedge b \neq 0 \wedge a > b \implies a - b \geq 0 \wedge b \geq 0$$
$$a \geq 0 \wedge b \geq 0 \wedge a \neq 0 \wedge b \neq 0 \wedge a > b \wedge b \geq 0 \wedge a - b \geq 0 \wedge \underset{k}{\exists}\, b = k \cdot t_1$$
$$\implies \underset{k}{\exists}\, a = k \cdot t_1$$
$$a \geq 0 \wedge b \geq 0 \wedge a \neq 0 \wedge b \neq 0 \wedge a \leq b \implies a \geq 0 \wedge b - a \geq 0$$
$$a \geq 0 \wedge b \geq 0 \wedge a \neq 0 \wedge b \neq 0 \wedge a \leq b \wedge a \geq 0 \wedge b - a \geq 0 \wedge \underset{k}{\exists}\, a = k \cdot t_2$$
$$\implies \underset{k}{\exists}\, a = k \cdot t_2$$
$$a \geq 0 \wedge b \geq 0 \wedge a \neq 0 \wedge b = 0 \implies \underset{k}{\exists}\, a = k \cdot a$$

The formulas above are universally quantified over the variables a, b, t_1, and t_2. The variables t_1 and t_2 were introduced for replacing the recursive call GCD on $a - b$, b, respectively a, $b - a$, because we do dot allow occurrences of the GCD in the verification conditions.
The termination condition is 2.5.

3. Synthesizing Optimal Algorithms. Case Study: Square Root

3.1. Program Synthesis meets Program Verification

Automated program synthesis is a difficult task and has been considered for a long time intractable [23]. Thus, has received little attention. With the advance of automated tools for software verification, it came into the attention of researchers, especially because of the benefits which it brings to program development. One of the most important benefits is that the automatically synthesized program is *correct-by-construction*. *Correct-by-construction paradigm* [24] was proposed by Edsger W. Dijkstra in 1970s. Given a mathematical specification of what a program is supposed to do, one applies mathematical transformations to the specification until it is turned into a program that can be executed.

It is well-known that in static program analysis the following are crucial: *i*) partial correctness, *ii*) termination, *iii*) complexity. We encode the synthesis problem into a program verification problem, namely, given a program specification and a program schema, we synthesize programs with the properties *i)* and *ii)*, having complexity at most *iii)*.

We applied this encoding to the problem of synthesizing reliable/optimal numeric algorithms. As a case study, we studied the problem of synthesizing optimal algorithms for computing the square root of a real number. More precisely, given the real number x and the error bound ε, we are searching for a real interval such that it contains $\sqrt{x}$ and its width is less than ε. We fix the algorithm schema, namely, iterative refining: the algorithm starts with an initial interval and repeatedly updates it by applying a refinement map, say R, on it until it becomes narrow enough. The

Algorithm 4 Algorithm Schema: Square Root Computation by Iterative Refining

<u>in</u> x, ε <u>reals</u> where $x > 1$, $\varepsilon > 0$
<u>out</u> $I = [L, U]$, <u>interval</u> where $\sqrt{x} \in I$ and $width(I) \leq \varepsilon$
$I \leftarrow [1, x]$
<u>while</u> $width(I) > \varepsilon$ <u>do</u>
    $I \leftarrow R(I, x)$
<u>return</u>$[I]$

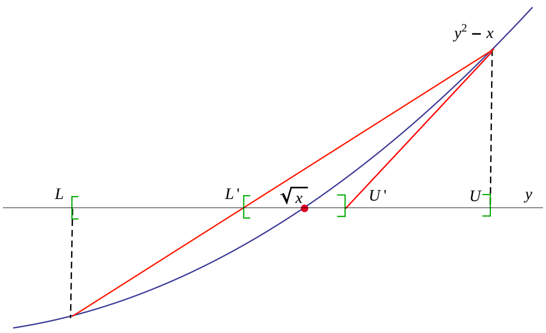
synthesis amounts to finding a refinement map R that ensures that the algorithm is *partially correct*, *terminates*, and *optimal*. All these can be formulated as *quantifier elimination (QE) problems over real numbers*. Hence, in principle, they can be carried out automatically. However the computational requirement is so huge, making the automatic synthesis practically impossible with the current general QE software. Hence, we performed some hand derivations and were able to synthesize semi-automatically optimal algorithms under suitable assumptions.

Motivating Example. As a motivating example, we considered the well-known refining map Secant-Newton, which is obtained by combining the secant map and the Newton map where the secant/Newton map is used for determining the lower/upper bound of the refined interval, that is,

$$R: \quad [L, U], x \quad \mapsto \quad \left[\frac{LU + x}{L + U}, \frac{U^2 + x}{2U}\right]$$

which can be easily derived from Figure 3.1. In the following, we formulate the notions

Figure 3.1.: Derivation of Secant-Newton Refinement Map



of partial correctness, termination, and complexity of Algorithm 4 and exemplify it on Secant-Newton refinement map.

Let $LoopInv(L, U, x) :\iff 0 < L \leq \sqrt{x} \leq U$ be the loop invariant of Algorithm 4. Partial correctness reduces basically to the proof that the loop invariant is inductively preserved by the execution of loop body, that is

$$\underset{L,U,x}{\forall} LoopInv(L, U, x) \Longrightarrow LoopInv(R(L, U)). \tag{3.1}$$

Specializing R to Secant-Newton refinement map and applying QE software (`Reduce` command of *Mathematica* [92]), we obtained that (3.1) is *True*. Hence, the Secant-Newton algorithm is partially correct.

Proving termination of Algorithm 4 reduces to the proof of termination of the loop. One of the most well-known techniques for proving that a loop terminates is to synthesize functions with the range into a well-founded set, called ranking functions or termination terms. Let $d(L,U) := U-L$ be the termination term. Then $\geq_\varepsilon$ defined as $x \geq_\varepsilon y := x \geq y+\varepsilon \ \wedge \ \varepsilon > 0$ is obviously a well-founded relation over $\mathbb{R}$. However, this approach is not suitable for our problem since is dependent on ε. An alternative is to show that $d(L,U)$ is a contraction map, that is, the following holds:

$$\underset{c\in(0,1)}{\exists} \text{ such that } c = \min_{p,q} \sup_{\substack{L,U,x \\ 0<L<\sqrt{x}<U}} \frac{d(R(L,U))}{d(L,U)}, \tag{3.2}$$

where c is the so-called Lipschitz constant.

Specializing R to Secant-Newton refinement map, and using constraint optimization techniques available in *Mathematica* (`MaxValue` command), we have found $c = \frac{1}{2}$. Hence the Secant-Newton algorithm terminates.

Computing the complexity of Algorithm 4 amounts to find the number of loop iterations, since operations like addition, multiplication, assignment and comparison over with real numbers require constant running time.

Lemma 3.1. *The number of loop iterations n of Algorithm 4 is given by*

$$n = \left\lceil \frac{\log_2 \frac{x-1}{\varepsilon}}{\log_2 \frac{1}{c}} \right\rceil$$

where

$$\underset{c\in(0,1)}{\exists} \textit{ such that } c = \min_{p,q} \sup_{\substack{L,U,x \\ 0<L<\sqrt{x}<U}} \frac{d(R(L,U))}{d(L,U)}.$$

Proof. Consider the estimate of $d(L,U)$ at each loop iteration as follows.

# iter	$d(L,U)$
0	$\leq \quad U-L$
1	$\leq c \cdot (U-L)$
2	$\leq c^2 \cdot (U-L)$
...	...
n	$\leq c^n \cdot (U-L)$

Moreover, at iteration n we know that $c^n \cdot (U-L) \leq \varepsilon$, hence $n \leq \log_c \frac{\varepsilon}{x-1}$. This estimate of n is not convenient since one can not figure out how c influences the value of n. By basis transformation we obtain $n = \left\lceil \frac{\log_2 \frac{\varepsilon}{x-1}}{\log_2 c} \right\rceil$ and further $n = \left\lceil \frac{\log_2 \frac{x-1}{\varepsilon}}{\log_2 \frac{1}{c}} \right\rceil$. $\square$

Remark 3.2. Note that, by Lemma 3.1, a small c gives a low number of loop iterations.

3.2. Program Synthesis as a QE Problem

Knowing the complexity of Secant-Newton algorithm, we are asking ourselves: *Is there any refinement map which is better than Secant-Newton?* In order to answer the question rigorously, one first needs to fix a search space, that is, a family of maps in which we search for a better map. We observe that the Secant-Newton refinement map is made of two rational functions of degree 2, where the numerator/the denominator is degree 2/degree 1 in the end of points, L and U, of the interval. These suggest the following choice of a search space: the family of all the maps with the form

$$
\begin{aligned}
R:\ & [L,U], x \mapsto [L', U'] \\
L' &= \frac{p_0 L^2 + p_1 LU + p_2 U^2 + x}{p_3 L + p_4 U} \\
U' &= \frac{q_0 L^2 + q_1 LU + q_2 U^2 + x}{q_3 L + q_4 U}
\end{aligned}
\tag{3.3}
$$

Then synthesizing optimal algorithms can be formulated as a constrained optimization problem as follows

$$
\min_{\substack{p,q \\ C(p,q)}} E(p,q), \tag{3.4}
$$

where

$$
\begin{aligned}
p &:= (p_0, p_1, p_2, p_3, p_4) \\
q &:= (q_0, q_1, q_2, q_3, q_4) \\
C(p,q) : \Longleftrightarrow & \quad \forall_{L,U,x}\ 0 < L \leq \sqrt{x} \leq U \Longrightarrow 0 < L' \leq \sqrt{x} \leq U' \\
E(p,q) := & \sup_{\substack{L,U,x \\ 0<L<\sqrt{x}<U}} \frac{U' - L'}{U - L}.
\end{aligned}
$$

The quantifier-free formula equivalent to $C(p,q)$ gives the values of p and q for which the algorithm is partially correct. The quantifier-free formula equivalent to $E(p,q)$ is a piecewise defined function which characterizes the values of p and q for which the algorithm terminates.

In principle, problem (3.4) could be solved by Algorithm 5. However, this is impossible due to the high computational complexity of general methods for QE: at steps 1 and 2 of Algorithm 5 we have to find the quantifier-free equivalent of a formula with three bound and ten free variables. In order to make the problem amenable to be solved semi-automatically, we consider that the refinement map fulfills additional natural assumptions. These assumptions are used for formulas simplification and variables reduction, hence they ease the task of software for QE in finding the quantifier-free equivalent. Synthesis of optimal algorithms under natural assumptions is investigated in Sections 3.4, 3.5, and, respectively, 3.6.

Algorithm 5 Synthesis Algorithm

<u>`in`</u>: $R,\ C(p,q),\ E(p,q)$, where

$$\begin{aligned}
&R : [L,U],\ x \mapsto [L',U'] \\
&L' = \tfrac{p_0L^2+p_1LU+p_2U^2+x}{p_3L+p_4U} \\
&U' = \tfrac{q_0L^2+q_1LU+q_2U^2+x}{q_3L+q_4U} \\
&C(p,q) :\iff \underset{L,U,x}{\forall}\ 0 < L \le \sqrt{x} \le U \Longrightarrow 0 < L' \le \sqrt{x} \le U' \\
&E(p,q) := \sup_{\substack{L,U,x \\ 0<L<\sqrt{x}<U}} \frac{U'-L'}{U-L}
\end{aligned}$$

<u>`out`</u>:

- $c = \min\limits_{\substack{p,q \\ C(p,q)}} E(p,q)$
- $C'(p,q) \iff (C(p,q) \Longrightarrow (E(p,q) = c))$

such that

1. $E(p,q) \ge c$
2. $E(p,q) = c \quad \iff \quad C'(p,q)$

Step 1. Eliminate $\forall$ from $C(p,q)$ and bring the result into the following form:

$$C(p,q) \iff \bigvee_i C_i$$

$C_i(p,q)$ – a conjunction of equations/inequalities in p,q

Step 2. Eliminate *sup* from $E(p,q)$ and bring the result into the following form:

$$E(p,q) = \begin{cases} \cdots & \cdots & \cdots \\ E_j(p,q) & \text{if} & G_j(p,q) \\ \cdots & \cdots & \cdots \end{cases}$$

$E_j(p,q)$ – an expression in p,q

$G_j(p,q)$ – a conjunction of equations/inequalities in p,q

Step 3. Let $V_{ij} = \min\limits_i \min\limits_j \min\limits_{C_i(p,q)\wedge G_j(p,q)} E_j(p,q)$. Using standard optimization technique, we determine V_{ij} and $C'_{ij}(p,q) \iff (C_i(p,q) \Longrightarrow (E(p,q) = V_{ij}))$ for each i, j. We find the minimum among V_{ij} for each i,j. For those i,j which give the minimum V_{ij}, we compute $C'(p,q) = \bigvee\limits_{i,j} C'_{ij}(p,q)$.

3.3. QE by CAD

In this section, we present QE by cylindrical algebraic decomposition (CAD) in the theory of reals, decidability results and complexity. We mainly follow the textbook [91].

3.3.1. The QE Problem and Applications

The *theory of reals* $T_{\mathbb{R}}$ has signature

$$\Sigma_{\mathbb{R}} : \{0, 1, +, -, \cdot, =, \geq\} ,$$

where

- 0 and 1 are constants,
- $+$ (addition) and $\cdot$ (multiplication) are binary functions,
- $-$ (negation) is a unary function, and
- $=$ (equality) and $\geq$ (weak inequality) are binary predicates.

$T_{\mathbb{R}}$ has a complex axiomatization. It contains axioms of *i*) abelian groups, *ii*) rings, *iii*) fields, *iv*) total orders, and *v*) real closed fields. (See [9] for a complete list.)

The problem of *QE for the theory of reals* can be stated as follows. *Given a formula in prenex normal form, find a quantifier-free formula equivalent to it.*

As an example, the problem of finding when a quadratic equation has a real root can be stated as a QE problem over reals as follows.

$$F \;:\Longleftrightarrow\; \underset{x}{\exists}\, ax^2 + bc + c = 0$$

Then a quantifier-free formula equivalent to F is F', where

$$F' \;:\Longleftrightarrow\; (a = 0 \land b \neq 0) \;\lor\; (a = 0 \land c = 0) \;\lor\; \big(a \neq 0 \land b^2 - 4ac \geq 0\big) .$$

QE has many applications: control [85], theorem proving in real geometry [26], program verification [49], numerical analysis [81], just to name a few.

3.3.2. A Brief Summary of QE Methods

Tarski [84] gave an algorithm for solving the QE problem for the theory of reals, hence he proved that it is decidable. Subsequently, Seidenberg [78] and Cohen [16] proposed other methods for the decidability result. However, their approaches have non-elementary complexity. This was until G. E. Collins [17] discovered the CAD algorithm, a completely new approach for QE, which has elementary complexity. More exactly, given a formula F, the time complexity of the method is $(mn)^{k^r} d^k$, where r is the number of free and bound variables in F, m is the number of polynomials occurring in F, n is the maximum degree of any polynomial in F, d is the maximum length of any integer coefficient of any polynomial in F, and k is some constant [17].

Since the discovery of the CAD algorithm, other methods for solving the QE have been proposed by Renegar [74] and Heintz *et. al.* [38] which are doubly exponential in the number of quantifier alternations. Weispfenning [89] discovered a QE algorithms based on comprehensive Gröbner bases without giving any complexity details.

Other research has focused on interesting fragments of the full theory. For example, Weispfenning considered formulas in which the bound variables appear only linearly [87] and at most cubically [88]. Hong [42] considered input of the form

$$\underset{x}{\exists}\, ax^2 + bx + c = 0 \ \wedge \ F,$$

where F is a quantifier-free formula. To be noted that Weispfenning method (so-called “virtual substitution”) is not based on CAD and solved many problems which CAD could not. That is because the complexity of the method is independent on the number of free variables in a formula.

It has been proved that the QE problem is inherently doubly exponential in the number of variables [20, 30]. Despite this, the QE based on CAD solved many non-trivial problems, either by formulating the problems in certain fragments and applying dedicated methods or by improving the CAD method. Speed-up of CAD method was accomplished by using improved projection operators [40, 59, 60] and/or developing advanced root isolation methods [43, 83]

The most well-known implementations of the CAD method are *QEPCAD-B* [11] and `Reduce` command of *Mathematica* [92]. Virtual substitution method is implemented in *Redlog* [25].

Nowadays, the CAD method is adapted to satisfiability modulo theories technologies [21] and has been successfully used to industrial software verification.

3.3.3. The Principles of QE by CAD

The idea of QE by CAD is to divide the n - dimensional space $\mathbb{R}^n$, where n is the number of variables in the given formula F, into areas for which the validity of F can be established by evaluating it at certain points. Checking the validity of F by simply inspecting it at certain points is possible due to the special type of decomposition of $\mathbb{R}^n$ (*cylindrical algebraic*) which is performed in the projection phase. Hence, in the *projection phase* the CAD of the free-variables space has the property that the formula F is sign-invariant in every cell of the decomposition. In the *stack construction (lifting)* phase an explicit representation of this decomposition is built. This decomposition can be used to determine the truth value of F in each cell of the decomposition. In the *formula construction* phase the decomposition is used to construct the free-variables formula F' equivalent to F.

3.3.4. What is CAD?

Let

$$F \quad :\Longleftrightarrow \quad (Q_1 x_{k+1})(Q_2 x_{k+2})...(Q_{r-k} x_r)\ F'(x_1, ..., x_r),$$

where $Q_i \in \{\forall, \exists\}$ and F' is a quantifier-free formula. F must be in prenex normal form.

An *algebraic decomposition* is one in each each cell is a semi-algebraic set. A *CAD* is an algebraic decomposition which has "cylinder" structure which will be explained in the following. For $\mathbb{R}$, any algebraic decomposition into open intervals and single points is a CAD: $c_1 = (-\infty, \alpha_1), c_2 = [\alpha_1, \alpha_1], c_3 = (\alpha_1, \alpha_2), ..., c_{2m} = [\alpha_m, \alpha_m], c_{2m+1} = (\alpha_m, \infty)$, where α_i are algebraic numbers and fulfill $<$ ordering on $\mathbb{R}$.

For $\mathbb{R}^n$, $n \geq 2$, the CAD is defined inductively as follows. A *stack* over the connected region $\mathcal{A}$ of $\mathbb{R}^i$ $(i = 2..n)$ is a decomposition of $\mathcal{A} \times \mathbb{R}$ into cells $c_1, ..., c_{2m+1}$ such that for any $\alpha \in \mathcal{A}$ the intersection of c_i with $\alpha \times \mathbb{R}$ is a CAD of $\mathbb{R}$ with the property that the cells in stack have the same nice ordering as for $\mathbb{R}$, that is $c_i \cap (\alpha \times \mathbb{R})$ is less than $c_j \cap (\alpha \times \mathbb{R})$ iff $i < j$. The even-indexed cells are called *sections* and are always single points, while the odd-indexed cells are called *sectors*. Hence, an algebraic decomposition D of $\mathbb{R}^{i+1}$ with the properties:

1. there exists a CAD of $\mathbb{R}^i$, say D' such that for any cell $c \in D$ there exists a cell $c' \in D'$ such that the projection onto $\mathbb{R}^i$ of c is c' (D' is called *induced CAD* of $\mathbb{R}^i$)
2. for the cell c' in the induced CAD of $\mathbb{R}^i$, the cells is D whose projections onto R^i are c' form a stack over c'.

is called a CAD.

The cell c' is called parent, the cells in the stack over c' are called children of c'.

3.3.5. Projection

Let A be the set of polynomials which appear in the formula F. Projection phase produces a set P (called *projection factor set*), $A \subseteq P \subset \mathbb{R}[x_1, ..., x_n]$ such that the decomposition defined by P is a CAD. In one step, projection produces a set $A^{n-1} = proj(A)$ (projection of A), $n \geq 2$, of polynomials in $n-1$ variables with the property that for every $proj(A)$ - invariant CAD D' of $\mathbb{R}^{n-1}$ there is an A - invariant CAD D of $\mathbb{R}^n$ that induces D', that is D' can be extended to a CAD of $\mathbb{R}^n$. A decomposition D of $\mathbb{R}^n$ is A - invariant iff every polynomial $p \in A$ is sign-invariant on every cell of D. Projection is applied recursively until univariate polynomials are obtained, that is it constructs the sets $A^n = A$, $A^{n-1} = proj(A)$, ..., A^1 of polynomials in n, $n-1$, ..., respectively, 1 variables. In the base case, the A^1 - invariant CAD of $\mathbb{R}$ is obtained. Because of the inductive nature of CAD's, an A^i - invariant CAD D^i is extended to an A^{i+1} - invariant CAD D^{i+1}, $1 \leq i < n$.

The set P computed in the projection phase is not unique. There have been proposed several projection operators [17, 40, 59]. Evidently, the size of the set P plays an important role for the speed of QE process.

3.3.6. Stack Construction

Let $p \in \mathbb{R}[x_1, ..., x_n]$. Then the *level* of p is the largest j such that $deg_{x_j}(p) > 0$. For $P \subseteq \mathbb{R}[x_1, ..., x_n]$, P_i is the set of polynomials in P with level i.

The stack construction phase as described in Arnon *at. al.* [3] constructs a sequence of CADs:

$$\begin{array}{lll} C_1 & - & \text{a CAD of } \mathbb{R} \text{ defined by } P_1 \\ C_2 & - & \text{a CAD of } \mathbb{R}^2 \text{ defined by } P_1 \cup P_2 \\ ... & & \\ C_{n-1} & - & \text{a CAD of } \mathbb{R}^{n-1} \text{ defined by } P_1 \cup P_2 \cup ... \cup P_{n-1} \\ C_n & - & \text{a CAD of } \mathbb{R}^n \text{ defined by } P_1 \cup P_2 \cup ... \cup P_n \end{array}$$

The CAD C_1 is used at the construction of C_2, C_2 is used at the construction of C_3, etc. C_1 is obtained by isolating the real roots of univariate polynomials. For each cell of the CAD of $\mathbb{R}$, one evaluates the polynomials in A^2 at a sample point and isolates their real roots, from which one produces a *stack over the cell.* Continuing in this manner, one finally obtains a CAD of R^n. Assuming that c_{i-1} is a cell in the CAD of $\mathbb{R}^{i-1}$ and $s = (s_1, ..., s_{i-1})$ is its sample point. The children of c_{i-1} will inherit its sample point, that is the sample point of the children of c_{i-1} have the first $i-1$ coordinates s. The sample point at level i can be found by root isolation and refinement.

3.3.7. Formula Construction

In this phase, the CAD of free-variables space and the truth value of F in each of these cells of the CAD is used to construct the quantifier-free formula equivalent to F. This method requires an *augmented projection* [17] which unfortunately increases the time required by projection and stack construction phases. Formula construction which does not require augmented projection was proposed by Hong [41]. However, his method does not work in all the cases. On the contrary, Brown [10] proposes formula construction method which general and quite effective in producing simple formula quickly.

3.4. Optimality of Secant-Newton Refinement Map

In this section, based on the observations from the Figure 3.1, we assume that the refinement map R defined in (3.3) is also contracting, that is

$$L \leq L' \leq \sqrt{x} \leq U' \leq U,$$

which we will call *contracting quadratic* maps. By choosing the values for the parameters $p = (p_0, \dots, p_4)$ and $q = (q_0, \dots, q_4)$, we get each member of the family. For instance, Secant-Newton map can be obtained by setting $p = (0, 1, 0, 1, 1)$ and $q = (0, 0, 1, 0, 2)$.

Using this assumption, we prove that Secant-Newton map is the *optimal* among all the contracting quadratic maps. By optimal, we mean that the output interval of Secant-Newton map is always proper subset of that of all the other contracting quadratic map, as long as $\sqrt{x}$ resides in the interior of the input interval.

It is important to note that the interval Newton map [36,66,73], without intersecting with the input interval, is not contracting. Hence, it does not belong to the family of maps that we consider in this section. Of course, one could turn any refinement map into a contracting one, simply by intersecting the output with the input interval. Using this approach, one could enlarge the family of maps so as to include the interval Newton map. Finding the optimal one among the enlarged family would be a natural extension to the work reported here and will be investigated in Section 3.6.

3.4.1. Main Result

In the following, we state precisely the main result. For this, we recall a few notations and notions.

Definition 3.3 (Quadratic refinement map)**.** *We say that a refinement map*

$$R : \; [L, U], x \;\; \mapsto \;\; [L', U']$$

is quadratic *if it has the following form*

$$\begin{aligned} L' &= \frac{p_0 L^2 + p_1 LU + p_2 U^2 + x}{p_3 L + p_4 U} \\ U' &= \frac{q_0 L^2 + q_1 LU + q_2 U^2 + x}{q_3 L + q_4 U}. \end{aligned}$$

We will denote it by $R_{p,q}$.

Definition 3.4 (Secant-Newton map)**.** *The Secant-Newton map is the quadratic refinement map R_{p^*,q^*} where $p^* = (0, 1, 0, 1, 1)$ and $q^* = (0, 0, 1, 0, 2)$, namely*

$$R_{p^*,q^*} : \; [L, U], x \;\; \mapsto \;\; [L^*, U^*]$$

where

$$L^* = \frac{LU + x}{L + U} = L + \frac{x - L^2}{L + U}$$
$$U^* = \frac{U^2 + x}{2U} = U + \frac{x - U^2}{2U}$$

Definition 3.5 (Contracting map). *We say that a map*

$$R : \ [L, U], x \ \mapsto \ [L', U']$$

is contracting *if*

$$\underset{L,U,x}{\forall} \ 0 < L \leq \sqrt{x} \leq U \implies L \leq L' \leq \sqrt{x} \leq U' \leq U. \tag{3.5}$$

Now we are ready to state the main result of the section.

Theorem 3.6 (Main Result). *Let $R_{p,q}$ be a contracting quadratic map which is not R_{p^*,q^*} (Secant-Newton). Then we have*

(a) $\underset{L,U,x}{\forall} \quad 0 < L \leq \sqrt{x} \leq U \quad \implies \quad R_{p^*,q^*}([L,U],x) \ \subseteq \ R_{p,q}([L,U],x)$

(b) $\underset{L,U,x}{\forall} \quad 0 < L < \sqrt{x} < U \quad \implies \quad R_{p^*,q^*}([L,U],x) \ \subsetneq \ R_{p,q}([L,U],x)$

Remark 3.7. It is important to pay a careful attention to a subtle difference between the two claims (a) and (b). In the first claim, $\sqrt{x}$ is allowed to lie on the boundary of the input interval, namely $\sqrt{x} = L$ or $\sqrt{x} = U$. In the second claim, $\sqrt{x}$ is required to lie in the interior of the input interval.

Remark 3.8. The first claim states that Secant-Newton map is never worse than any other contracting quadratic map as along as $\sqrt{x}$ resides in the input interval. The second claim states that Secant-Newton map is always better than all the other contracting quadratic maps as long as $\sqrt{x}$ resides in the interior of the input interval.

3.4.2. Proof

In this section, we prove the main result (Theorem 3.6). For the sake of easy readability, the proof will be divided into several lemmas, which are interesting on their own. The main theorem follows immediately from the Lemmas 3.12 and 3.13.

Lemma 3.9. *Let $R_{p,q}$ be a contracting quadratic map. Then we have*

$$\begin{array}{lllllll} 0 & = & p_0 - p_3 + 1 & = & p_1 - p_4 & = & p_2 \\ 0 & = & q_2 - q_4 + 1 & = & q_1 - q_3 & = & q_0. \end{array}$$

Proof. Let $R_{p,q}$ be a contracting quadratic map. Then p, q satisfy the condition (3.5). The proof essentially consist of instantiating the condition (3.5) on $x = L^2$ and $x = U^2$.

By instantiating the condition (3.5) with $x = L^2$ and recalling the definition of L', we have

$$\underset{L,U}{\forall}\ 0 < L \leq U \implies \frac{p_0 L^2 + p_1 LU + p_2 U^2 + L^2}{p_3 L + p_4 U} = L.$$

By removing the denominator and collecting, we have

$$\underset{L,U}{\forall}\ (L, U) \in D \implies g\,(L, U) = 0,$$

where

$$\begin{aligned} D &= \{(L, U) : 0 < L \leq U\}, \\ g\,(L, U) &= (p_0 - p_3 + 1)L^2 + (p_1 - p_4)LU + p_2 U^2. \end{aligned}$$

Since the bivariate polynomial g is zero over the 2-dim real domain D, it must be identically zero. Thus its coefficients $p_0 - p_3 + 1,\ p_1 - p_4,\ p_2$ must be all zero.

By instantiating the condition (3.5) with $x = U^2$ and recalling the definition of U', we have

$$\underset{L,U}{\forall}\ 0 < L \leq U \implies \frac{q_0 L^2 + q_1 LU + q_2 U^2 + U^2}{q_3 L + q_4 U} = U.$$

By removing the denominator and collecting, we have

$$\underset{L,U}{\forall}\ (L, U) \in D \implies g\,(L, U) = 0,$$

where

$$\begin{aligned} D &= \{(L, U) : 0 < L \leq U\}, \\ g\,(L, U) &= q_0 L^2 + (q_1 - q_3)LU + (q_2 - q_4 + 1)U^2. \end{aligned}$$

Since the bivariate polynomial g is zero over the 2-dim real domain D, it must be identically zero. Thus its coefficients $q_0,\ q_1 - q_3,\ q_2 - q_4 + 1$ must be all zero. $\square$

Lemma 3.10. *Let $R_{p,q}$ be a contracting quadratic map. Then we have*

$$\begin{aligned} L' &= L + \frac{x - L^2}{p_3 L + p_4 U} \\ U' &= U + \frac{x - U^2}{q_3 L + q_4 U}. \end{aligned}$$

Proof. Let $R_{p,q}$ be a contracting quadratic map. From Lemma 3.9, we have

$$0 = p_0 - p_3 + 1 = p_1 - p_4 = p_2$$
$$0 = q_2 - q_4 + 1 = q_1 - q_3 = q_0.$$

Recalling the definition of L' and U', we have

$$L' := \frac{(p_3-1)L^2 + p_4 LU + x}{p_3 L + p_4 U}$$
$$U' := \frac{q_3 LU + (q_4-1)U^2 + x}{q_3 L + q_4 U}.$$

By simplifying we have

$$L' = L + \frac{x - L^2}{p_3 L + p_4 U}$$
$$U' = U + \frac{x - U^2}{q_3 L + q_4 U}.$$

□

Lemma 3.11. *Let $R_{p,q}$ be a contracting quadratic map. Then we have*

$$p_3 + p_4 - 2 \geq 0 \qquad p_4 - 1 \geq 0$$
$$q_3 + q_4 - 2 \geq 0 \qquad q_4 - 2 \geq 0.$$

Proof. Let $R_{p,q}$ be a contracting quadratic map. Using Lemma 3.10, we can rewrite the condition (3.5) as

$$\underset{L,U,x}{\forall} \quad 0 < L \leq \sqrt{x} \leq U \quad \Longrightarrow \quad L \leq L + \frac{x - L^2}{p_3 L + p_4 U} \leq \sqrt{x} \leq U + \frac{x - U^2}{q_3 L + q_4 U} \leq U.$$

Simplifying, we have

$$\underset{L,U,x}{\forall} \quad 0 < L \leq \sqrt{x} \leq U \quad \Longrightarrow \quad \begin{array}{l} 0 \leq \frac{(\sqrt{x}-L)(\sqrt{x}+L)}{p_3 L + p_4 U} \leq \sqrt{x} - L \\ \wedge \\ 0 \leq \frac{(U-\sqrt{x})(U+\sqrt{x})}{q_3 L + q_4 U} \leq U - \sqrt{x}. \end{array}$$

By restricting the universal quantification to $\sqrt{x} \neq L$ and $\sqrt{x} \neq U$, we have

$$\underset{L,U,x}{\forall} \quad 0 < L < \sqrt{x} < U \quad \Longrightarrow \quad \begin{array}{l} 0 \leq \frac{\sqrt{x}+L}{p_3 L + p_4 U} \leq 1 \\ \wedge \\ 0 \leq \frac{\sqrt{x}+U}{q_3 L + q_4 U} \leq 1. \end{array}$$

By canceling the denominators, we have

$$\underset{L,U,x}{\forall} \quad 0 < L < \sqrt{x} < U \quad \Longrightarrow \quad \begin{array}{l} 0 \leq \sqrt{x} + L \leq p_3 L + p_4 U \\ \wedge \\ 0 \leq \sqrt{x} + U \leq q_3 L + q_4 U. \end{array}$$

By rewriting it, we have

$$\underset{L,U,x}{\forall} \quad 0 < L < \sqrt{x} < U \Longrightarrow \begin{array}{l} 0 \leq (p_3 + p_4 - 2)\, L + (p_4 - 1)\,(U - L) + (U - \sqrt{x}) \\ \wedge \\ 0 \leq (q_3 + q_4 - 2)\, L + (q_4 - 2)\,(U - L) + (U - \sqrt{x})\,. \end{array} \tag{3.6}$$

Claim: $p_3 + p_4 - 2 \geq 0$. Assume otherwise, that is, $p_3 + p_4 - 2 < 0$. We will show that it contradicts (3.6). Let

$$L = 1 + \frac{|2p_4 - 1|}{-\,(p_3 + p_4 - 2)},\ U = L + 2,\ x = (U - 1)^2$$

Then obviously $0 < L < \sqrt{x} < U$. However

$$\begin{aligned}
&(p_3 + p_4 - 2)\, L + (p_4 - 1)\,(U - L) + \left(U - \sqrt{x}\right) \\
&= (p_3 + p - 2) \left(1 + \frac{|2p_4 - 1|}{-\,(p_3 + p_4 - 2)}\right) + (p_4 - 1)\, 2 + 1 \\
&= (p_3 + p_4 - 2) - |2p_4 - 1| + 2p_4 - 1 \\
&\leq p_3 + p_4 - 2 \\
&< 0
\end{aligned}$$

contradicting (3.6).

Claim: $q_3 + q_4 - 2 \geq 0$. Assume otherwise, that is, $q_3 + q_4 - 2 < 0$. We will show that it contradicts (3.6). Let

$$L = 1 + \frac{|2q_4 - 3|}{-\,(q_3 + q_4 - 2)},\ U = L + 2,\ x = (U - 1)^2\,.$$

Then obviously $0 < L < \sqrt{x} < U$. However

$$\begin{aligned}
&(q_3 + q_4 - 2)\, L + (q_4 - 2)\,(U - L) + \left(U - \sqrt{x}\right) \\
&= (q_3 + q_4 - 2) \left(1 + \frac{|2q_4 - 3|}{-\,(q_3 + q_4 - 2)}\right) + (q_4 - 2)\, 2 + 1 \\
&= (q_3 + q_4 - 2) - |2q_4 - 3| + 2q_4 - 3 \\
&\leq q_3 + q_4 - 2 \\
&< 0
\end{aligned}$$

contradicting (3.6).

Claim: $p_4 - 1 \geq 0$. Assume otherwise, that is, $p_4 - 1 < 0$. We will show that it contradicts (3.6). Let

$$L = 1,\ U = 3 + \frac{|p_3 + p_4 - 1|}{-(p_4 - 1)},\ x = (U - 1)^2 .$$

Then obviously $0 < L < \sqrt{x} < U$. However

$$\begin{aligned}
&(p_3 + p_4 - 2)\, L + (p_4 - 1)\,(U - L) + \left(U - \sqrt{x}\right) \\
&= p_3 + p_4 - 2 + (p_4 - 1)\left(2 + \frac{|p_3 + p_4 - 1|}{-(p_4 - 1)}\right) + 1 \\
&= p_3 + p_4 - 1 + 2(p_4 - 1) - |p_3 + p_4 - 1| \\
&\leq 2(p_4 - 1) \\
&< 0
\end{aligned}$$

contradicting (3.6).

Claim: $q_4 - 2 \geq 0$. Assume otherwise, that is, $q_4 - 2 < 0$. We will show that it contradicts (3.6). Let

$$L = 1,\ U = 3 + \frac{|q_3 + q_4 - 1|}{-(q_4 - 2)},\ x = (U - 1)^2 \ .$$

Then obviously $0 < L < \sqrt{x} < U$. However

$$\begin{aligned}
&(q_3 + q_4 - 2)\, L + (q_4 - 2)\,(U - L) + \left(U - \sqrt{x}\right) \\
&= q_3 + q_4 - 2 + (q_4 - 2)\left(2 + \frac{|q_3 + q_4 - 1|}{-(q_4 - 2)}\right) + 1 \\
&= q_3 + q_4 - 1 + 2\,(q_4 - 2) - |q_3 + q_4 - 1| \\
&\leq 2\,(q_4 - 2) \\
&< 0
\end{aligned}$$

contradicting (3.6). □

Now we are ready to prove the two claims in Main Theorem. The following lemma (Lemma 3.12) will prove the claim (a) and the subsequent lemma (Lemma 3.13) will prove the claim (b).

Lemma 3.12 (Main Theorem (a)). *Let $R_{p,q}$ be a contracting quadratic map which is not R_{p^*,q^*} (Secant-Newton). Then we have*

$$\underset{L,U,x}{\forall} \quad 0 < L \leq \sqrt{x} \leq U \quad \Longrightarrow \quad R_{p^*,q^*}([L,U],x) \ \subseteq\ R_{p,q}([L,U],x)$$

Proof. Let $R_{p,q}$ be a contracting quadratic map which is not R_{p^*,q^*} (Secant-Newton), that is, $p \neq p^*$ or $q \neq q^*$. Let L, U, x be arbitrary such that $0 < L \leq \sqrt{x} \leq U$. We need to show

$$R_{p^*,q^*}([L,U],x) \quad \subseteq \quad R_{p,q}([L,U],x)$$

Note

$$\begin{aligned}
& R_{p^*,q^*}([L,U],x) \quad \subseteq \quad R_{p,q}([L,U],x) \\
\iff & L' \leq L^* \ \wedge \ U^* \leq U' \\
\iff & L + \tfrac{x-L^2}{p_3 L + p_4 U} \leq L + \tfrac{x-L^2}{L+U} \\
& \wedge \qquad\qquad\qquad\qquad\qquad\qquad \text{(Due to Lemma 3.10)} \\
& U + \tfrac{x-U^2}{2U} \leq U + \tfrac{x-U^2}{q_3 L + q_4 U} \\
\iff & \left(x - L^2\right)\left(\tfrac{1}{L+U} - \tfrac{1}{p_3 L + p_4 U}\right) \geq 0 \\
& \wedge \\
& \left(U^2 - x\right)\left(\tfrac{1}{2U} - \tfrac{1}{q_3 L + q_4 U}\right) \geq 0 \\
\iff & \left(x - L^2\right)\left(\tfrac{1}{2L+(U-L)} - \tfrac{1}{(p_3+p_4)L + p_4(U-L)}\right) \geq 0 \\
& \wedge \\
& \left(U^2 - x\right)\left(\tfrac{1}{2L+2(U-L)} - \tfrac{1}{(q_3+q_4)L + q_4(U-L)}\right) \geq 0 \\
\iff & \left(x - L^2\right)\tfrac{(p_3+p_4-2)L + (p_4-1)(U-L)}{(2L+(U-L))((p_3+p_4)L + p_4(U-L))} \geq 0 \\
& \wedge \\
& \left(U^2 - x\right)\tfrac{(q_3+q_4-2)L + (q_4-2)(U-L)}{(2L+2(U-L))((q_3+q_4)L + q_4(U-L))} \geq 0 \\
\iff & \left(x - L^2\right)\left((p_3 + p_4 - 2)L + (p_4 - 1)\left(U - L\right)\right) \geq 0 \\
& \wedge \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(Due to Lemma 3.11)} \\
& \left(U^2 - x\right)\left((q_3 + q_4 - 2)L + (q_4 - 2)\left(U - L\right)\right) \geq 0 \\
\iff & \mathit{true}. \quad \text{(Due to Lemma 3.11)}
\end{aligned}$$

Main Theorem (a) has been proved. □

Lemma 3.13 (Main Theorem (b))**.** *Let $R_{p,q}$ be a contracting quadratic map which is not R_{p^*,q^*} (Secant-Newton). Then we have*

$$\underset{L,U,x}{\forall} \quad 0 < L < \sqrt{x} < U \quad \Longrightarrow \quad R_{p^*,q^*}([L,U],x) \quad \subsetneq \quad R_{p,q}([L,U],x)$$

Proof. Let $R_{p,q}$ be a contracting quadratic map which is not R_{p^*,q^*} (Secant-Newton), that is, $p \neq p^*$ or $q \neq q^*$. Let L, U, x be arbitrary such that $0 < L < \sqrt{x} < U$. We need to show

$$R_{p^*,q^*}([L,U],x) \quad \subsetneq \quad R_{p,q}([L,U],x)$$

Following a similar process as in the proof of Lemma 3.12, we have

$$
\begin{aligned}
& R_{p^*,q^*}([L,U],x) \;\subsetneq\; R_{p,q}([L,U],x) \\
\Longleftrightarrow\; & L' < L^* \;\vee\; U^* < U' \quad \text{(Due to Lemma 3.12)} \\
\Longleftrightarrow\; & L + \tfrac{x-L^2}{p_3L+p_4U} < L + \tfrac{x-L^2}{L+U} \\
& \vee \qquad\qquad\qquad\qquad\qquad \text{(Due to Lemma 3.10)} \\
& U + \tfrac{x-U^2}{2U} < U + \tfrac{x-U^2}{q_3L+q_4U} \\
\Longleftrightarrow\; & \tfrac{1}{L+U} - \tfrac{1}{p_3L+p_4U} > 0 \\
& \vee \qquad\qquad\qquad\qquad\qquad \text{(since } L < \sqrt{x} < U\text{)} \\
& \tfrac{1}{2U} - \tfrac{1}{q_3L+q_4U} > 0 \\
\Longleftrightarrow\; & \tfrac{1}{2L+(U-L)} - \tfrac{1}{(p_3+p_4)L+p_4(U-L)} > 0 \\
& \vee \\
& \tfrac{1}{2L+2(U-L)} - \tfrac{1}{(q_3+q_4)L+q_4(U-L)} > 0 \\
\Longleftrightarrow\; & \tfrac{(p_3+p_4-2)L+(p_4-1)(U-L)}{(2L+(U-L))((p_3+p_4)L+p_4(U-L))} > 0 \\
& \vee \\
& \tfrac{(q_3+q_4-2)L+(q_4-2)(U-L)}{(2L+2(U-L))((q_3+q_4)L+q_4(U-L))} > 0 \\
\Longleftrightarrow\; & (p_3+p_4-2)L + (p_4-1)\,(U-L) > 0 \\
& \vee \qquad\qquad\qquad\qquad\qquad \text{(Due to Lemma 3.11)} \\
& (q_3+q_4-2)L + (q_4-2)\,(U-L) > 0 \\
\Longleftrightarrow\; & p_3+p_4-2 \neq 0 \;\vee\; p_4-1 \neq 0 \\
& \vee \qquad\qquad\qquad\qquad\qquad \text{(Due to Lemma 3.11)} \\
& q_3+q_4-2 \neq 0 \;\vee\; q_4-2 \neq 0 \\
\Longleftrightarrow\; & \neg\,(p_3+p_4-2=0 \;\wedge\; p_4-1=0 \;\wedge q_3+q_4-2=0 \;\wedge\; q_4-2=0) \\
\Longleftrightarrow\; & \neg\,(p_3=1 \;\wedge\; p_4=1 \;\wedge q_3=0 \;\wedge\; q_4=2) \\
\Longleftrightarrow\; & \neg\,(p=p^* \wedge q=q^*) \quad \text{(Due to Lemma 3.9)} \\
\Longleftrightarrow\; & p \neq p^* \vee q \neq q^* \\
\Longleftrightarrow\; & \mathit{true}.
\end{aligned}
$$

Main Theorem (b) has been proved. □

3.5. The Complexity of Contracting Quadratic Maps

We proved in Lemma 3.1 that the number of loop iterations of Algorithm 4 depends on the value of the Lipschitz constant of the associated quadratic map.

In this section, we prove that the contracting quadratic maps have the best Lipschitz constant $\frac{1}{2}$, and give the values of the parameters p and q for which the best Lipschitz constant is attained.

3.5.1. Main Result

Before stating the main result of this section, let us recall the following definitions.

$$C(p,q) :\iff \underset{L,U,x}{\forall}\ 0 < L \le \sqrt{x} \le U \Longrightarrow 0 < L \le L' \le \sqrt{x} \le U' \le U$$
$$E(p,q) := \sup_{\substack{L,U,x \\ 0<L<\sqrt{x}<U}} \left[\frac{U'-L'}{U-L}\right]$$

Theorem 3.14. *Let $R_{p,q}$ be a contracting quadratic map. Then*
(a) $E(p,q) \ge \frac{1}{2}$
(b) $E(p,q) = \frac{1}{2} \iff$
$$\begin{array}{l} p_0 = p_3 - 1 \ \wedge\ p_1 = p_4 \ \wedge\ p_2 = 0 \\ \wedge \\ q_0 = 0 \ \wedge\ q_1 = q_3 \ \wedge\ q_2 = 1 \ \wedge\ q_4 = 2 \\ \wedge \\ 2 - p_4 \le p_3 \le 4 - p_4 \ \wedge\ 1 \le p_4 \le 2 \ \wedge\ 0 \le q_3 \le 2. \end{array}$$

3.5.2. Proof

Let p, q be arbitrary but fixed. Assume that $R_{p,q}$ is a contracting quadratic map. We need to show (a) and (b). The proof of (a) is immediate by Section 3.4 (Lemma 3.18). The proof of (b) is divided into lemmas, ending with Lemma 3.19 proving (b).

Before we plunge into the details of the proofs of (b), we note, from Lemmas 3.11 and 3.9, respectively, that the following hold

$$p_3 + p_4 \ge 2 \ \wedge\ p_4 \ge 1 \ \wedge\ q_3 + q_4 \ge 2 \ \wedge\ q_4 \ge 2.$$

and

$$\begin{array}{lll} p_0 = p_3 - 1 & p_1 = p_4 & p_2 = 0 \\ q_0 = 0 & q_1 = q_3 & q_2 = q_4 - 1 \end{array}$$

Hence from now on, we will replace all the $p_0, p_1, p_2, q_0, q_1, q_2$ by the corresponding left hand side expressions.
Recalling the definition of L' and U', we have

$$L' := \frac{(p_3-1)L^2 + p_4 LU + x}{p_3 L + p_4 U}$$
$$U' := \frac{q_3 LU + (q_4-1)U^2 + x}{q_3 L + q_4 U}.$$

We make the remark that many intermediary results leading to the final proof were obtained automatically using *Mathematica* computer algebra system. We do not list here all the intermediary results. The file containing all the necessary computations leading to the main result of this section can be found at `http://www.risc.jku.at/people/merascu/PhDThesis/SquareRoot/C-C.nb`.

In the following lemma we eliminate the quantifier from $E(p,q)$.

Lemma 3.15. *$E(p,q)$ is the same as the function in Appendix C.1.*

Proof. Note that

$$\begin{aligned}
E(p,q) &:= \sup_{\substack{L,U,x \\ 0<L<\sqrt{x}<U}} \left[\frac{\frac{q_3LU+(q_4-1)U^2+x}{q_3L+q_4U} - \frac{(p_3-1)L^2+p_4LU+x}{p_3L+p_4U}}{U-L}\right] \\
&= \sup_{\substack{L,U,x \\ 0<L<\sqrt{x}<U}} \left[\frac{\frac{q_3LU+(q_4-1)U^2}{q_3L+q_4U} - \frac{(p_3-1)L^2+p_4LU}{p_3L+p_4U} + \left(\frac{1}{p_3L+p_4U} - \frac{1}{p_3L+p_4U}\right)x}{U-L}\right] \\
&= \sup_{\substack{L,U,x \\ 0<L<\sqrt{x}<U}} \left[\frac{\frac{q_3LU+(q_4-1)U^2}{q_3L+q_4U} - \frac{(p_3-1)L^2+p_4LU}{p_3L+p_4U}}{U-L} + \frac{\frac{1}{q_3L+q_4U} - \frac{1}{p_3L+p_4U}}{U-L}x\right] \\
&= \sup\left\{E_1(p,q),\ E_2(p,q)\right\},
\end{aligned}$$

where

$$\begin{aligned}
E_1(p,q) &:= \sup_{\substack{L,U \\ 0<L<U \\ \frac{1}{q_3L+q_4U}\geq\frac{1}{p_3L+p_4U}}} \left[\frac{\frac{q_3LU+(q_4-1)U^2}{q_3L+q_4U} - \frac{(p_3-1)L^2+p_4LU}{p_3L+p_4U}}{U-L} + \frac{\frac{1}{q_3L+q_4U} - \frac{1}{p_3L+p_4U}}{U-L}U^2\right] \\
E_2(p,q) &:= \sup_{\substack{L,U \\ 0<L<U \\ \frac{1}{q_3L+q_4U}\leq\frac{1}{p_3L+p_4U}}} \left[\frac{\frac{q_3LU+(q_4-1)U^2}{q_3L+q_4U} - \frac{(p_3-1)L^2+p_4LU}{p_3L+p_4U}}{U-L} + \frac{\frac{1}{q_3L+q_4U} - \frac{1}{p_3L+p_4U}}{U-L}L^2\right]
\end{aligned}$$

By simplifying the above expressions, we have

$$\begin{aligned}
E_1(p,q) &= \sup_{\substack{L,U \\ 0<L<U \\ \frac{1}{q_3L+q_4U}\geq\frac{1}{p_3L+p_4U}}} \left[1 - \frac{L+U}{p_3L+p_4U}\right] \\
E_2(p,q) &= \sup_{\substack{L,U \\ 0<L<U \\ \frac{1}{q_3L+q_4U}\leq\frac{1}{p_3L+p_4U}}} \left[1 - \frac{L+U}{q_3L+q_4U}\right]
\end{aligned}$$

By Lemmas 3.16, respectively 3.17, we have

$$E_1(p,q) = \begin{cases} 1 + \frac{(p_3-p_4)-(q_3-q_4)}{p_4q_3-p_3q_4} & \text{if} \quad \sigma_1 > 0 \ \wedge \ \sigma_2 > 0 \ \wedge \ \sigma_3 > 0 \\ 1 - \frac{1}{p_4} & \text{if} \quad \sigma_1 > 0 \ \wedge \ \sigma_2 > 0 \ \wedge \ \sigma_3 \le 0 \\ 1 - \frac{2}{p_3+p_4} & \text{if} \quad \sigma_2 \le 0 \ \wedge \ \sigma_3 > 0 \\ 1 - \frac{1}{p_4} & \text{if} \quad \sigma_1 \ge 0 \ \wedge \ \sigma_2 \le 0 \ \wedge \ \sigma_3 \le 0 \\ 1 + \frac{(p_3-p_4)-(q_3-q_4)}{p_4q_3-p_3q_4} & \text{if} \quad \sigma_1 < 0 \ \wedge \ \sigma_2 \le 0 \ \wedge \ \sigma_3 \le 0 \\ -\infty & \text{if} \quad \sigma_1 \le 0 \ \wedge \ \sigma_2 > 0 \end{cases}$$

$$E_2(p,q) = \begin{cases} 1 + \frac{(q_3-q_4)-(p_3-p_4)}{q_4p_3-q_3p_4} & \text{if} \quad \sigma_1 < 0 \ \wedge \ \sigma_2 < 0 \ \wedge \ \sigma_4 > 0 \\ 1 - \frac{1}{q_4} & \text{if} \quad \sigma_1 < 0 \ \wedge \ \sigma_2 < 0 \ \wedge \ \sigma_4 \le 0 \\ 1 - \frac{2}{q_3+q_4} & \text{if} \quad \sigma_2 \ge 0 \ \wedge \ \sigma_4 > 0 \\ 1 - \frac{1}{q_4} & \text{if} \quad \sigma_1 \le 0 \ \wedge \ \sigma_2 \ge 0 \ \wedge \ \sigma_4 \le 0 \\ 1 + \frac{(q_3-q_4)-(p_3-p_4)}{q_4p_3-q_3p_4} & \text{if} \quad \sigma_1 > 0 \ \wedge \ \sigma_2 \ge 0 \ \wedge \ \sigma_4 \le 0 \\ -\infty & \text{if} \quad \sigma_1 \ge 0 \ \wedge \ \sigma_2 < 0 \end{cases}$$

where

$$\begin{aligned} \sigma_1 &:= p_4 - q_4 \\ \sigma_2 &:= q_3 + q_4 - (p_3 + p_4) \\ \sigma_3 &:= p_3 - p_4 \\ \sigma_4 &:= q_3 - q_4 \end{aligned}$$

In order to compute $E(p,q)$ we proceed as follows. Note that $E_1(p,q)$ and $E_2(p,q)$ have, respectively, the form

$$\begin{aligned} E_1(p,q) = &\quad v_{1i}(p,q) \quad \text{if} \quad c_{1i}(p,q), \qquad i = 1..n \\ E_2(p,q) = &\quad v_{2j}(p,q) \quad \text{if} \quad c_{2j}(p,q), \qquad j = 1..m \end{aligned}$$

Hence, $E(p,q)$ has the form

$$E(p,q) = \begin{cases} v_{1i}(p,q) & \text{if} \quad c_{1i}(p,q) \ \wedge \ c_{2j}(p,q) \ \wedge \ v_{1i}(p,q) \ge v_{2j}(p,q) \\ v_{2j}(p,q) & \text{if} \quad c_{1i}(p,q) \ \wedge \ c_{2j}(p,q) \ \wedge \ v_{1i}(p,q) < v_{2j}(p,q) \end{cases},$$

where $i = 1..n$, $j = 1..m$.
Finally, we combined

$$c_{1i}(p,q) \ \wedge \ c_{2j}(p,q) \ \wedge \ v_{1i}(p,q) \ge v_{2j}(p,q)$$

and, respectively,

$$c_{1i}(p,q) \ \wedge \ c_{2j}(p,q) \ \wedge \ v_{1i}(p,q) < v_{2j}(p,q),$$

where $i = 1..n$, $j = 1..m$ with

$$p_3 + p_4 \geq 2 \ \wedge \ p_4 \geq 1 \ \wedge \ q_3 + q_4 \geq 2 \ \wedge \ q_4 \geq 2.$$

The value of $E(p,q)$ is too lengthy to be added here and is listed in Appendix C.1. $\square$

Lemma 3.16. *Let*

$$E_1(p,q) := \sup_{\substack{L,U \\ 0<L<U \\ \frac{1}{q_3 L + q_4 U} \geq \frac{1}{p_3 L + p_4 U}}} \left[1 - \frac{L+U}{p_3 L + p_4 U}\right].$$

Then

$$E_1(p,q) = \begin{cases} 1 + \frac{(p_3-p_4)-(q_3-q_4)}{p_4 q_3 - p_3 q_4} & \textit{if} \quad \sigma_1 > 0 \ \wedge \ \sigma_2 > 0 \ \wedge \ \sigma_3 > 0 \\ 1 - \frac{1}{p_4} & \textit{if} \quad \sigma_1 > 0 \ \wedge \ \sigma_2 > 0 \ \wedge \ \sigma_3 \leq 0 \\ 1 - \frac{2}{p_3+p_4} & \textit{if} \quad \sigma_2 \leq 0 \ \wedge \ \sigma_3 > 0 \\ 1 - \frac{1}{p_4} & \textit{if} \quad \sigma_1 \geq 0 \ \wedge \ \sigma_2 \leq 0 \ \wedge \ \sigma_3 \leq 0 \\ 1 + \frac{(p_3-p_4)-(q_3-q_4)}{p_4 q_3 - p_3 q_4} & \textit{if} \quad \sigma_1 < 0 \ \wedge \ \sigma_2 \leq 0 \ \wedge \ \sigma_3 \leq 0 \\ -\infty & \textit{if} \quad \sigma_1 \leq 0 \ \wedge \ \sigma_2 > 0 \end{cases},$$

where

$$\begin{aligned} \sigma_1 &:= p_4 - q_4 \\ \sigma_2 &:= q_3 + q_4 - (p_3 + p_4) \\ \sigma_3 &:= p_3 - p_4 \end{aligned}$$

Proof. Let

$$E_1(p,q) := \sup_{\substack{L,U \\ 0<L<U \\ \frac{1}{q_3 L + q_4 U} \geq \frac{1}{p_3 L + p_4 U}}} \left[1 - \frac{L+U}{p_3 L + p_4 U}\right]$$

By Lemma 3.11, the following holds

$$p_3 + p_4 \geq 2 \ \wedge \ p_4 \geq 1 \ \wedge \ q_3 + q_4 \geq 2 \ \wedge \ q_4 \geq 2.$$

Dividing by $L > 0$, we have

$$E_1(p,q) = \sup_{\substack{L,U \\ 0<1<\frac{U}{L} \\ \frac{1}{q_3 + q_4 \frac{U}{L}} \geq \frac{1}{p_3 + p_4 \frac{U}{L}}}} \left[1 - \frac{1 + \frac{U}{L}}{p_3 + p_4 \frac{U}{L}}\right]$$

By renaming $\frac{U}{L}$ with K, and using the fact that $p_3 + p_4K > 0$ and $q_3 + q_4K > 0$, we have

$$
\begin{aligned}
E_1(p,q) &= \sup_{\substack{K \\ K>1 \\ \frac{1}{q_3+q_4K} \geq \frac{1}{p_3+p_4K}}} \left[1 - \frac{1+K}{p_3+p_4K}\right] \\
&= \sup_{\substack{K \\ K-1>0 \\ (p_4-q_4)(K-1)\geq q_3+q_4-(p_3+p_4)}} \left[1 - \frac{K-1+2}{p_3+p_4+p_4(K-1)}\right]
\end{aligned}
$$

By renaming $K-1$ with K, we have

$$
\begin{aligned}
E_1(p,q) = &\sup_{\substack{K \\ K>0 \\ (p_4-q_4)K\geq q_3+q_4-(p_3+p_4)}} \left[1 - \frac{K+2}{p_3+p_4+p_4K}\right] \\
= &\sup_{\substack{K \\ K>0 \\ (p_4-q_4)K\geq q_3+q_4-(p_3+p_4)}} \left[1 - \frac{\frac{1}{p_4}(p_3+p_4+p_4K) - \frac{1}{p_4}(p_3+p_4+p_4K) + 2 + K}{p_3+p_4+p_4c}\right] \\
= &\sup_{\substack{K \\ K>0 \\ (p_4-q_4)K\geq q_3+q_4-(p_3+p_4)}} \left[1 - \frac{1}{p_4} + \frac{1}{p_4^2}\frac{p_3-p_4}{K+\frac{p_3+p_4}{p_4}}\right] \\
= &\,1 - \frac{1}{p_4} + \frac{1}{p_4^2} \sup_{\substack{K \\ K>0 \\ (p_4-q_4)K\geq q_3+q_4-(p_3+p_4)}} \left[\frac{p_3-p_4}{K+\frac{p_3+p_4}{p_4}}\right]
\end{aligned}
$$

Let $\sigma_1 := p_4 - q_4$. Considering the signs of σ_1, we have

$$
E_1(p,q) = 1 - \frac{1}{p_4} + \frac{1}{p_4^2}
\begin{cases}
\sup_{\substack{K \\ K>0 \\ K\geq \frac{q_3+q_4-(p_3+p_4)}{\sigma_1}}} \left[\frac{p_3-p_4}{K+\frac{p_3+p_4}{p_4}}\right] & \text{if} \quad \sigma_1 > 0 \\
\sup_{\substack{K \\ K>0 \\ K\leq \frac{q_3+q_4-(p_3+p_4)}{\sigma_1}}} \left[\frac{p_3-p_4}{K+\frac{p_3+p_4}{p_4}}\right] & \text{if} \quad \sigma_1 < 0 \\
\sup_{\substack{K \\ K>0 \\ 0\geq q_3+q_4-(p_3+p_4)}} \left[\frac{p_3-p_4}{K+\frac{p_3+p_4}{p_4}}\right] & \text{if} \quad \sigma_1 = 0
\end{cases}
$$

Let $\sigma_2 := q_3 + q_4 - (p_3 + p_4)$. Considering the signs of σ_2, we have

$$E_1(p,q) = 1 - \frac{1}{p_4} + \frac{1}{p_4^2}\begin{cases} \sup\limits_{\substack{K \\ K \geq \frac{\sigma_2}{\sigma_1}}} \left[\frac{p_3 - p_4}{K + \frac{p_3+p_4}{p_4}}\right] & \text{if} \quad \sigma_1 > 0 \ \wedge \ \sigma_2 > 0 \\ \sup\limits_{\substack{K \\ K > 0}} \left[\frac{p_3 - p_4}{K + \frac{p_3+p_4}{p_4}}\right] & \text{if} \quad \sigma_1 > 0 \ \wedge \ \sigma_2 \leq 0 \\ -\infty & \text{if} \quad \sigma_1 < 0 \ \wedge \ \sigma_2 > 0 \\ \sup\limits_{\substack{K \\ 0 < K \leq \frac{\sigma_2}{\sigma_1}}} \left[\frac{p_3 - p_4}{K + \frac{p_3+p_4}{p_4}}\right] & \text{if} \quad \sigma_1 < 0 \ \wedge \ \sigma_2 \leq 0 \\ -\infty & \text{if} \quad \sigma_1 = 0 \ \wedge \ \sigma_2 > 0 \\ \sup\limits_{\substack{K \\ K > 0}} \left[\frac{p_3 - p_4}{K + \frac{p_3+p_4}{p_4}}\right] & \text{if} \quad \sigma_1 = 0 \ \wedge \ \sigma_2 \leq 0 \end{cases}$$

Combining the cases, we have

$$E_1(p,q) = 1 - \frac{1}{p_4} + \frac{1}{p_4^2}\begin{cases} \sup\limits_{\substack{K \\ K \geq \frac{\sigma_2}{\sigma_1}}} \left[\frac{p_3 - p_4}{K + \frac{p_3+p_4}{p_4}}\right] & \text{if} \quad \sigma_1 > 0 \ \wedge \ \sigma_2 > 0 \\ \sup\limits_{\substack{K \\ K > 0}} \left[\frac{p_3 - p_4}{K + \frac{p_3+p_4}{p_4}}\right] & \text{if} \quad \sigma_1 \geq 0 \ \wedge \ \sigma_2 \leq 0 \\ \sup\limits_{\substack{K \\ 0 < K \leq \frac{\sigma_2}{\sigma_1}}} \left[\frac{p_3 - p_4}{K + \frac{p_3+p_4}{p_4}}\right] & \text{if} \quad \sigma_1 < 0 \ \wedge \ \sigma_2 \leq 0 \\ -\infty & \text{if} \quad \sigma_1 \leq 0 \ \wedge \ \sigma_2 > 0 \end{cases}$$

Let $\sigma_3 := p_3 - p_4$. Considering the signs of σ_3, we have

$$E_1(p,q) = 1 - \frac{1}{p_4} + \frac{1}{p_4^2}\begin{cases} \sigma_3 \sup\limits_{\substack{K \\ K \geq \frac{\sigma_2}{\sigma_1}}} \left[\frac{1}{K+\frac{p_3+p_4}{p_4}}\right] & \text{if} \quad \sigma_1 > 0 \ \wedge\ \sigma_2 > 0 \ \wedge\ \sigma_3 > 0 \\ \sigma_3 \inf\limits_{\substack{K \\ K \geq \frac{\sigma_2}{\sigma_1}}} \left[\frac{1}{K+\frac{p_3+p_4}{p_4}}\right] & \text{if} \quad \sigma_1 > 0 \ \wedge\ \sigma_2 > 0 \ \wedge\ \sigma_3 \leq 0 \\ \sigma_3 \sup\limits_{\substack{K \\ K > 0}} \left[\frac{1}{K+\frac{p_3+p_4}{p_4}}\right] & \text{if} \quad \sigma_1 \geq 0 \ \wedge\ \sigma_2 \leq 0 \ \wedge\ \sigma_3 > 0 \\ \sigma_3 \inf\limits_{\substack{K \\ K > 0}} \left[\frac{1}{K+\frac{p_3+p_4}{p_4}}\right] & \text{if} \quad \sigma_1 \geq 0 \ \wedge\ \sigma_2 \leq 0 \ \wedge\ \sigma_3 \leq 0 \\ \sigma_3 \sup\limits_{\substack{K \\ 0 < K \leq \frac{\sigma_2}{\sigma_1}}} \left[\frac{1}{K+\frac{p_3+p_4}{p_4}}\right] & \text{if} \quad \sigma_1 < 0 \ \wedge\ \sigma_2 \leq 0 \ \wedge\ \sigma_3 > 0 \\ \sigma_3 \inf\limits_{\substack{K \\ 0 < K \leq \frac{\sigma_2}{\sigma_1}}} \left[\frac{1}{K+\frac{p_3+p_4}{p_4}}\right] & \text{if} \quad \sigma_1 < 0 \ \wedge\ \sigma_2 \leq 0 \ \wedge\ \sigma_3 \leq 0 \\ -\infty & \text{if} \quad \sigma_1 \leq 0 \ \wedge\ \sigma_2 > 0 \end{cases}$$

Since $\frac{p_3+p_4}{p_4} > 0$ and $\frac{\sigma_2}{\sigma_1} \geq 0$ (from the side conditions when it appears), we have

$$E_1(p,q) = 1 - \frac{1}{p_4} + \frac{1}{p_4^2}\begin{cases} \sigma_3 \frac{1}{\frac{\sigma_2}{\sigma_1}+\frac{p_3+p_4}{p_4}} & \text{if} \quad \sigma_1 > 0 \ \wedge\ \sigma_2 > 0 \ \wedge\ \sigma_3 > 0 \\ 0 & \text{if} \quad \sigma_1 > 0 \ \wedge\ \sigma_2 > 0 \ \wedge\ \sigma_3 \leq 0 \\ \sigma_3 \frac{1}{\frac{p_3+p_4}{p_4}} & \text{if} \quad \sigma_1 \geq 0 \ \wedge\ \sigma_2 \leq 0 \ \wedge\ \sigma_3 > 0 \\ 0 & \text{if} \quad \sigma_1 \geq 0 \ \wedge\ \sigma_2 \leq 0 \ \wedge\ \sigma_3 \leq 0 \\ \sigma_3 \frac{1}{\frac{p_3+p_4}{p_4}} & \text{if} \quad \sigma_1 < 0 \ \wedge\ \sigma_2 \leq 0 \ \wedge\ \sigma_3 > 0 \\ \sigma_3 \frac{1}{\frac{\sigma_2}{\sigma_1}+\frac{p_3+p_4}{p_4}} & \text{if} \quad \sigma_1 < 0 \ \wedge\ \sigma_2 \leq 0 \ \wedge\ \sigma_3 \leq 0 \\ -\infty & \text{if} \quad \sigma_1 \leq 0 \ \wedge\ \sigma_2 > 0 \end{cases}$$

By combining, we have

$$E_1(p,q) = 1 - \frac{1}{p_4} + \frac{1}{p_4^2}\begin{cases} \sigma_3 \frac{1}{\frac{\sigma_2}{\sigma_1}+\frac{p_3+p_4}{p_4}} & \text{if} \quad \sigma_1 > 0 \ \wedge\ \sigma_2 > 0 \ \wedge\ \sigma_3 > 0 \\ 0 & \text{if} \quad \sigma_1 > 0 \ \wedge\ \sigma_2 > 0 \ \wedge\ \sigma_3 \leq 0 \\ \sigma_3 \frac{1}{\frac{p_3+p_4}{p_4}} & \text{if} \quad \sigma_2 \leq 0 \ \wedge\ \sigma_3 > 0 \\ 0 & \text{if} \quad \sigma_1 \geq 0 \ \wedge\ \sigma_2 \leq 0 \ \wedge\ \sigma_3 \leq 0 \\ \sigma_3 \frac{1}{\frac{\sigma_2}{\sigma_1}+\frac{p_3+p_4}{p_4}} & \text{if} \quad \sigma_1 < 0 \ \wedge\ \sigma_2 \leq 0 \ \wedge\ \sigma_3 \leq 0 \\ -\infty & \text{if} \quad \sigma_1 \leq 0 \ \wedge\ \sigma_2 > 0 \end{cases}$$

By simplifying the values, we have

$$E_1(p,q) = \begin{cases} 1 + \frac{(p_3-p_4)-(q_3-q_4)}{p_4q_3-p_3q_4} & \text{if} \quad \sigma_1 > 0 \quad \wedge \quad \sigma_2 > 0 \quad \wedge \quad \sigma_3 > 0 \\ 1 - \frac{1}{p_4} & \text{if} \quad \sigma_1 > 0 \quad \wedge \quad \sigma_2 > 0 \quad \wedge \quad \sigma_3 \le 0 \\ 1 - \frac{2}{p_3+p_4} & \text{if} \quad \quad \sigma_2 \le 0 \quad \wedge \quad \sigma_3 > 0 \\ 1 - \frac{1}{p_4} & \text{if} \quad \sigma_1 \ge 0 \quad \wedge \quad \sigma_2 \le 0 \quad \wedge \quad \sigma_3 \le 0 \\ 1 + \frac{(p_3-p_4)-(q_3-q_4)}{p_4q_3-p_3q_4} & \text{if} \quad \sigma_1 < 0 \quad \wedge \quad \sigma_2 \le 0 \quad \wedge \quad \sigma_3 \le 0 \\ -\infty & \text{if} \quad \sigma_1 \le 0 \quad \wedge \quad \sigma_2 > 0 \end{cases}$$

where

$$\begin{aligned} \sigma_1 &= p_4 - q_4 \\ \sigma_2 &= q_3 + q_4 - (p_3 + p_4) \\ \sigma_3 &= p_3 - p_4 \end{aligned}$$

□

Lemma 3.17. *Let*

$$E_2(p,q) := \sup_{\substack{L,U \\ 0<L<U \\ \frac{1}{q_3L+q_4U} \le \frac{1}{p_3L+p_4U}}} \left[1 - \frac{L+U}{q_3L+q_4U}.\right]$$

Then we have

$$E_2(p,q) = \begin{cases} 1 + \frac{(q_3-q_4)-(p_3-p_4)}{q_4p_3-q_3p_4} & \textit{if} \quad \sigma_1 < 0 \quad \wedge \quad \sigma_2 < 0 \quad \wedge \quad \sigma_4 > 0 \\ 1 - \frac{1}{q_4} & \textit{if} \quad \sigma_1 < 0 \quad \wedge \quad \sigma_2 < 0 \quad \wedge \quad \sigma_4 \le 0 \\ 1 - \frac{2}{q_3+q_4} & \textit{if} \quad \phantom{\sigma_1 < 0 \quad \wedge} \quad \sigma_2 \ge 0 \quad \wedge \quad \sigma_4 > 0 \\ 1 - \frac{1}{q_4} & \textit{if} \quad \sigma_1 \le 0 \quad \wedge \quad \sigma_2 \ge 0 \quad \wedge \quad \sigma_4 \le 0 \\ 1 + \frac{(q_3-q_4)-(p_3-p_4)}{q_4p_3-q_3p_4} & \textit{if} \quad \sigma_1 > 0 \quad \wedge \quad \sigma_2 \ge 0 \quad \wedge \quad \sigma_4 \le 0 \\ -\infty & \textit{if} \quad \sigma_1 \ge 0 \quad \wedge \quad \sigma_2 < 0 \end{cases},$$

where

$$\begin{aligned} \sigma_1 &:= p_4 - q_4 \\ \sigma_2 &:= q_3 + q_4 - (p_3 + p_4) \\ \sigma_3 &:= q_3 - q_4 \end{aligned}$$

Proof. Let

$$E_2(p,q) := \sup_{\substack{L,U \\ 0<L<U \\ \frac{1}{q_3L+q_4U} \le \frac{1}{p_3L+p_4U}}} \left[1 - \frac{L+U}{q_3L+q_4U}\right].$$

Note that $E_2(p,q)$ can be obtained from $E_1(p,q)$ defined in Lemma 3.17 by swapping p_3, p_4 with q_3, q_4, respectively. Hence,

$$E_2(p,q) = \begin{cases} 1 + \frac{(q_3-q_4)-(p_3-p_4)}{q_4p_3-q_3p_4} & \text{if} \quad \sigma_1 < 0 \;\wedge\; \sigma_2 < 0 \;\wedge\; \sigma_3 > 0 \\ 1 - \frac{1}{q_4} & \text{if} \quad \sigma_1 < 0 \;\wedge\; \sigma_2 < 0 \;\wedge\; \sigma_3 \leq 0 \\ 1 - \frac{2}{q_3+q_4} & \text{if} \quad \sigma_2 \geq 0 \;\wedge\; \sigma_3 > 0 \\ 1 - \frac{1}{q_4} & \text{if} \quad \sigma_1 \leq 0 \;\wedge\; \sigma_2 \geq 0 \;\wedge\; \sigma_3 \leq 0 \\ 1 + \frac{(q_3-q_4)-(p_3-p_4)}{q_4p_3-q_3p_4} & \text{if} \quad \sigma_1 > 0 \;\wedge\; \sigma_2 \geq 0 \;\wedge\; \sigma_3 \leq 0 \\ -\infty & \text{if} \quad \sigma_1 \geq 0 \;\wedge\; \sigma_2 < 0 \end{cases}$$

where

$$\begin{aligned} \sigma_1 &:= p_4 - q_4 \\ \sigma_2 &:= q_3 + q_4 - (p_3 + p_4) \\ \sigma_3 &:= q_3 - q_4 \end{aligned}$$

□

Lemma 3.18 (Main Theorem (a))**.** *We have* $E(p,q) \geq \frac{1}{2}$.

Proof. The proof is immediate, since $C(p^*,q^*)$ is true and $E(p^*,q^*) = \frac{1}{2}$, where $p^* = (0,1,0,1,1)$, $q^* = (0,0,1,0,2)$. □

Lemma 3.19 (Main Theorem (b))**.** *We have*

$$E(p,q) = \frac{1}{2} \iff \begin{array}{l} p_0 = p_3 - 1 \;\wedge\; p_1 = p_4 \;\wedge\; p_2 = 0 \\ \wedge \\ q_0 = 0 \;\wedge\; q_1 = q_3 \;\wedge\; q_2 = 1 \;\wedge\; q_4 = 2 \\ \wedge \\ 2 - p_4 \leq p_3 \leq 4 - p_4 \;\wedge\; 1 \leq p_4 \leq 2 \;\wedge\; 0 \leq q_3 \leq 2. \end{array}$$

Proof. In order prove the claim, we have to solve the following constrained optimization problem:

$$\min_{\substack{p,q \\ C(p,q)}} E(p,q).$$

By Lemma 3.15, the standard optimization problem can be brought into the following form:

$$\min_i \min_{C(p,q) \wedge G_i(p,q)} E_i(p,q),$$

where

$$\begin{aligned} C(p,q) &\iff p_3 + p_4 \geq 2 \;\wedge\; p_4 \geq 1 \;\wedge\; q_3 + q_4 \geq 2 \;\wedge\; q_4 \geq 2 \\ G_i(p,q) &- \text{a conjunction of equations/inequalities in } p, q \\ E_i(p,q) &- \text{an expression in } p, q \end{aligned}$$

The values of $G_i(p,q)$ and $E_i(p,q)$, $i = 1..n$, are listed in Appendix C.

Further, we need to solve the following standard optimization problems:

$$\text{Minimize } E_i(p,q) \text{ subject to } C(p,q) \wedge G_i(p,q), \text{ for each i.} \tag{3.7}$$

These were carried out by symbolic constrained optimization (`Minimize`) available in *Mathematica*. The routine is listed in Appendix C.2 and has the following specification

- Input: list of the form $\{\{e_1, c_1\}, ..., \{e_n, c_n\}\}$, where e_i is an expression in p, q, c_i is a conjunction of equalities/inequalities in p, q, $i = 1..n$;
- Output: list of the form $\{\{\{v_1, s_1\}, C_1\}, ..., \{\{v_n, s_n\}, C_n\}\}$, where v_i is the minimum value in the region determined by c_i, s_i is a substitution for p, q for which $e_i = v_i$, C_i is a disjunction of conjunctions of equalities/inequalities in p, q for which $e_i = v_i$, $i = 1..n$.

The output is listed in Appendix C.3. Finally, from the list in Appendix C.3, we take the minimum v, that is $\frac{1}{2}$, and the disjunction of all C_i which give v, that is

$$1 \leq p_4 \leq 2 \;\wedge\; 2 - p_4 \leq p_3 \leq 4 - p_4 \;\wedge\; 0 \leq q_3 \leq 2 \;\wedge\; q_4 = 2.$$

□

3.6. Towards Optimal Square Root Algorithms

We proved in Theorem 3.14 that the contracting quadratic maps have the best Lipschitz constant $\frac{1}{2}$. A natural question arises: *Are there any quadratic maps which have a smaller Lipschitz constant?* To answer this question, we dropped off the contraction condition on quadratic maps, but we imposed the following natural assumption on them

$$C(p,q) \;:\Longleftrightarrow\; C_1(p,q) \;\wedge\; C_2(p) \;\wedge\; C_3(q),$$

where

$$\begin{aligned} C_1(p,q) :\Longleftrightarrow\;& \underset{L,U,x}{\forall}\, 0 < L \leq \sqrt{x} \leq U \implies 0 < L' \leq \sqrt{x} \leq U' \\ C_2(p) :\Longleftrightarrow\;& 0 \leq p_4 \leq 2 \;\wedge\; p_3 + p_4 = 2 \;\wedge\; p_0 + p_1 + p_2 = 1 \\ C_3(q) :\Longleftrightarrow\;& 0 \leq q_4 \leq 2 \;\wedge\; q_3 + q_4 = 2 \;\wedge\; q_0 + q_1 + q_2 = 1 \end{aligned}$$

3.6.1. Main Result

Before stating the main result, let us recall the following definitions.

$$
\begin{aligned}
C(p,q) :\Longleftrightarrow\ & C_1(p,q) \wedge C_2(p) \ \wedge\ C_3(q) \\
C_1(p,q) :\Longleftrightarrow\ & \underset{L,U,x}{\forall}\ 0 < L \le \sqrt{x} \le U \implies 0 < L' \le \sqrt{x} \le U' \\
C_2(p) :\Longleftrightarrow\ & 0 \le p_4 \le 2 \ \wedge\ p_3 + p_4 = 2 \ \wedge\ p_0 + p_1 + p_2 = 1 \\
C_3(q) :\Longleftrightarrow\ & 0 \le q_4 \le 2 \ \wedge\ q_3 + q_4 = 2 \ \wedge\ q_0 + q_1 + q_2 = 1 \\
C(p,q) :\Longleftrightarrow\ & C_1(p,q) \ \wedge\ C_2(p) \ \wedge\ C_3(q) \\
E(p,q) :=\ & \sup_{\substack{L,U,x \\ 0<L<\sqrt{x}<U}} \frac{U' - L'}{U - L}
\end{aligned}
$$

Theorem 3.20 (Main Theorem)**.** *Let $R_{p,q}$ be a quadratic map satisfying $C(p,q)$. Then*
(a) $E(p,q) \ge \frac{1}{4}$

(b) $E(p,q) = \frac{1}{4} \iff$
$$
\begin{aligned}
& p_0 = 1 - p_1 - p_2 \ \wedge\ p_3 = 2 - p_4 \\
& \wedge \\
& q_0 = 1 - q_1 - q_2 \ \wedge\ q_3 = 2 - q_4 \\
& \wedge \\
& q_1 \le p_1 \le 1 \ \wedge\ p_2 = 0 \ \wedge\ p_4 = 1 \\
& \wedge \\
& \tfrac{1}{2} \le q_1 \le 1 \ \wedge\ q_2 = \tfrac{1}{4} \ \wedge\ q_4 = 1
\end{aligned}
$$

3.6.2. Proof

Let p,q be arbitrary but fixed. Assume that $R_{p,q}$ is a quadratic map satisfying $C(p,q)$. We need to show (a) and (b). The proofs for (a) and (b) will be divided into lemmas, ending with Lemma 3.29 proving (b) and Lemma 3.30 proving (a).

Before we plunge into the details of the proofs of (a) and (b), we note, from C_2 and C_3, that:

$$
\begin{aligned}
p_0 &= 1 - p_1 - p_2 \qquad & p_3 &= 2 - p_4 \\
q_0 &= 1 - q_1 - q_2 \qquad & q_3 &= 2 - q_4.
\end{aligned}
$$

Hence from now on, we will replace all the p_0,q_0,p_3,q_3 by the corresponding left hand side expressions.

Recalling the definitions of p, q, $C_2(p)$, $C_3(q)$, L', respectively U', we have

$$\begin{aligned}
p &= (p_1, p_2, p_4)\\
q &= (q_1, q_2, q_4)\\
C_2\,(p) &\iff 0 \le p_4 \le 2\\
C_3\,(q) &\iff 0 \le q_4 \le 2\\
L' &= \frac{(1-p_1-p_2)L^2 + p_1 LU + p_2 U^2 + x}{(2-p_4)L + p_4 U}\\
U' &= \frac{(1-q_1-q_2)L^2 + q_1 LU + q_2 U^2 + x}{(2-q_4)L + q_4 U}
\end{aligned}$$

Many results used in the following were obtained automatically either by using *QEPCAD-B* (elimination of universal quantification) or by using *Mathematica* computer algebra system (constrained optimization). Due to their length, they are not listed here. The file containing all the necessary computations leading to the main result of this section can be found at `http://www.risc.jku.at/people/merascu/PhDThesis/SquareRoot/PI-PI.nb`.

In the following we eliminate the universal quantification from $C(p, q)$.

Lemma 3.21. *We have*

$$\begin{aligned}
&1 \le p_4 \le 2 \;\wedge\; p_1 \ge 0 \;\wedge\; p_2 = 0 \;\wedge\; p_1 - p_4 \le 0\\
&\wedge\\
&0 \le q_4 \le 2 \;\wedge\; q_4^2 - 4q_2 \le 0 \;\wedge\; q_1 + 2q_2 - q_4 \ge 0.
\end{aligned}$$

Proof. The proof follows immediately by combining Lemmas 3.22 and 3.24, and Lemmas 3.23 and 3.25.

By combining Lemmas 3.22 and 3.24, we have

$$\begin{aligned}
&0 \le p_4 \le 2\\
&\wedge\\
&p_2 \ge 0 \wedge \big(2p_2 + p_1 - 2 > 0 \vee p_1 \ge 0 \vee (p_1 + 2p_2)^2 - 8p_2 < 0\big)\\
&\wedge\\
&p_2 \le 0 \wedge p_4 - p_2 - 1 \ge 0 \wedge p_4 - 2p_2 - p_1 \ge 0\\
\iff\; &1 \le p_4 \le 2 \;\wedge\; p_1 \ge 0 \;\wedge\; p_2 = 0 \;\wedge\; p_1 - p_4 \le 0.
\end{aligned}$$

By combining Lemmas 3.23 and 3.25, we have

$$\begin{aligned}
&0 \le q_4 \le 2\\
&\wedge\\
&q_2 \ge 0 \wedge (2q_2 + q_1 - 2 > 0 \vee q_1 \ge 0 \vee (q_1 + 2q_2)^2 - 8q_2 < 0)\\
&\wedge\\
&4q_2 - q_4^2 \ge 0 \wedge q_4 - 2q_2 - q_1 \le 0\\
\iff\; &0 \le q_4 \le 2 \;\wedge\; q_4^2 - 4q_2 \le 0 \;\wedge\; q_1 + 2q_2 - q_4 \ge 0.
\end{aligned}$$

The simplification of the formulas above was done by *QEPCAD-B* [11]. □

Note that the following lemmas solve quantifier elimination problems. Due to their complexity, none of the state-of-the-art solvers [11,25,92] was able to deliver a quantifier-free equivalent directly. Hence, we performed variable elimination by exploiting equality constraints and properties of monotonic functions on an interval.

Lemma 3.22. *We have*

$$(A) \iff (B)\,,$$

where

$$\begin{aligned}
(A) :\iff & \underset{L,U,x}{\forall}\ 0 < L \leq \sqrt{x} \leq U \implies L' > 0\\
(B) :\iff & p_2 \geq 0 \ \wedge\ (p_1 \geq 0 \ \vee\ p_1 + 2p_2 - 2 > 0 \ \vee\ (p_1 + 2p_2)^2 - 8p_2 < 0).
\end{aligned}$$

Proof. We have

$$\begin{aligned}
& \underset{L,U,x}{\forall}\ 0 < L \leq \sqrt{x} \leq U \implies L' > 0\\
\iff & \underset{L,U,x}{\forall}\ 0 < L \leq \sqrt{x} \leq U \implies \frac{(1-p_1-p_2)L^2 + p_1LU + p_2U^2 + x}{(2-p_4)L + p_4U} > 0\\
\iff & \underset{L,U,x}{\forall}\ 0 < L \leq \sqrt{x} \leq U \implies (1-p_1-p_2)L^2 + p_1LU + p_2U^2 + x > 0.
\end{aligned}$$

In the above we used the fact that

$$C_2(p) \ \wedge\ 0 < L \leq U \implies 2L + p_4(U - L) > 0. \tag{3.8}$$

Note that the function

$$f_1(\sqrt{x}) := (\sqrt{x})^2 + (1-p_1-p_2)L^2 + p_1LU + p_2U^2$$

is convex on $[L, U]$. Hence, we have

$$\begin{aligned}
& \underset{L,U,x}{\forall}\ 0 < L \leq \sqrt{x} \leq U \implies f_1(\sqrt{x}) > 0\\
\iff & \underset{L,U,x}{\forall}\ 0 < L \leq \sqrt{x} \leq U \implies f_1(L) > 0\\
\iff & \underset{L,U}{\forall}\ 0 < L \leq U \implies (1-p_1-p_2)L^2 + p_1LU + p_2U^2 + L^2 > 0\\
\iff & \underset{L,U}{\forall}\ 0 < L \leq U \implies (2-p_1-p_2)L^2 + p_1LU + p_2U^2 > 0.
\end{aligned}$$

By using *QEPCAD-B*, we obtained

$$\begin{aligned}
& \underset{L,U}{\forall}\ 0 < L \leq U \implies (2-p_1-p_2)L^2 + p_1LU + p_2U^2 > 0\\
\iff & p_2 \geq 0 \ \wedge\ (p_1 \geq 0 \ \vee\ p_1 + 2p_2 - 2 > 0 \ \vee\ (p_1 + 2p_2)^2 - 8p_2 < 0).
\end{aligned}$$

□

Lemma 3.23. *We have*

$$(A) \iff (B)\,,$$

where

$$\begin{aligned}
(A) :\iff & \underset{L,U,x}{\forall}\ 0 < L \leq \sqrt{x} \leq U \implies U' > 0 \\
(B) :\iff & q_2 \geq 0 \ \wedge\ (q_1 \geq 0 \ \vee\ q_1 + 2q_2 - 2 > 0 \ \vee\ (q_1 + 2q_2)^2 - 8q_2 < 0).
\end{aligned}$$

Proof. Immediate by replacing p with q in Lemma 3.22. □

Lemma 3.24. *We have*

$$(A) \iff (B)\,,$$

where

$$\begin{aligned}
(A) :\iff & \underset{L,U,x}{\forall}\ 0 < L \ \leq \sqrt{x} \leq U \implies L' \leq \sqrt{x} \\
(B) :\iff & p_2 \leq 0 \ \wedge\ p_2 - p_4 + 1 \leq 0 \ \wedge\ p_1 + 2p_2 - p_4 \leq 0.
\end{aligned}$$

Proof. We have

$$\begin{aligned}
& \underset{L,U,x}{\forall}\ 0 < L \ \leq \sqrt{x} \leq U \implies L' \leq \sqrt{x} \\
\iff & \underset{L,U,x}{\forall}\ 0 < L \ \leq \sqrt{x} \leq U \implies \frac{(1-p_1-p_2)L^2 + p_1 LU + p_2 U^2 + x}{(2-p_4)L + p_4 U} \leq \sqrt{x} \\
\iff & \underset{L,U,x}{\forall}\ 0 < L \ \leq \sqrt{x} \leq U \\
& \implies x - ((2-p_4)L + p_4 U)\sqrt{x} + (1-p_1-p_2)L^2 + p_1 LU + p_2 U^2 \leq 0.
\end{aligned}$$

In the above we used (3.8).
Note that the function

$$f_2(\sqrt{x}) := \left(\sqrt{x}\right)^2 - ((2-p_4)L + p_4 U)\sqrt{x} + (1-p_1-p_2)L^2 + p_1 LU + p_2 U^2,$$

is convex on $[L, U]$. Hence, we have

$$
\begin{aligned}
& \underset{L,U,x}{\forall}\ 0 < L \ \leq \sqrt{x} \leq U \implies f_2\left(\sqrt{x}\right) \leq 0 \\
\iff\ & \underset{L,U}{\forall}\ 0 < L \ \leq U \implies f_2\left(L\right) \leq 0 \\
& \wedge \\
& \underset{L,U}{\forall}\ 0 < L \ \leq U \implies f_2\left(U\right) \leq 0 \\
\iff\ & \underset{L,U}{\forall}\ 0 < L \ \leq U \\
& \implies L^2 - L((2-p_4)L + p_4 U) + (1 - p_1 - p_2)L^2 + p_1 LU + p_2 U^2 \leq 0 \\
& \wedge \\
& \underset{L,U}{\forall}\ 0 < L \ \leq U \\
& \implies U^2 - U((2-p_4)L + p_4 U) + (1 - p_1 - p_2)L^2 + p_1 LU + p_2 U^2 \leq 0 \\
\iff\ & \underset{L,U}{\forall}\ 0 < L \ \leq U \implies (U-L)\left(L\left(p_1 + 2p_2 - p_4\right) + (U-L)\, p_2\right) \leq 0 \\
& \wedge \\
& \underset{L,U}{\forall}\ 0 < L \ \leq U \implies (U-L)\left(L\left(p_1 + 2p_2 - p_4\right) + (U-L)\left(p_2 - p_4 + 1\right)\right) \leq 0.
\end{aligned}
$$

By using *QEPCAD-B*, we obtained

$$
\begin{aligned}
& \underset{L,U}{\forall}\ 0 < L \ \leq U \implies (U-L)\left(L\left(p_1 + 2p_2 - p_4\right) + (U-L)\, p_2\right) \leq 0 \\
& \wedge \\
& \underset{L,U}{\forall}\ 0 < L \ \leq U \implies (U-L)\left(L\left(p_1 + 2p_2 - p_4\right) + (U-L)\left(p_2 - p_4 + 1\right)\right) \leq 0 \\
\iff\ & p_1 + 2p_2 - p_4 \leq 0 \wedge p_2 \leq 0 \\
& \wedge \\
& p_1 + 2p_2 - p_4 \leq 0 \wedge p_2 - p_4 + 1 \leq 0 \\
\iff\ & p_2 \leq 0 \ \wedge\ p_2 - p_4 + 1 \leq 0 \ \wedge\ p_1 + 2p_2 - p_4 \leq 0.
\end{aligned}
$$

□

Lemma 3.25. *We have*

$$(A) \iff (B)\,,$$

where

$$
\begin{aligned}
(A) :\iff\ & \underset{L,U,x}{\forall}\ 0 < L \ \leq \sqrt{x} \leq U \implies \sqrt{x} \leq U' \\
(B) :\iff\ & q_4^2 - 4q_2 \leq 0 \ \wedge\ q_1 + 2q_2 - q_4 \geq 0.
\end{aligned}
$$

Proof. We have

$$
\begin{aligned}
& \underset{L,U,x}{\forall}\ 0 < L \ \le \sqrt{x} \le U \implies \sqrt{x} \le U' \\
\iff & \underset{L,U,x}{\forall}\ 0 < L \ \le \sqrt{x} \le U \implies \sqrt{x} \le \frac{(1-q_1-q_2)L^2 + q_1LU + q_2U^2 + x}{(2-q_4)L + q_4U} \\
\iff & \underset{L,U,x}{\forall}\ 0 < L \ \le \sqrt{x} \le U \\
& \implies x - ((2-q_4)L + q_4U)\sqrt{x} + (1-q_1-q_2)L^2 + q_1LU + q_2U^2 \ge 0.
\end{aligned}
$$

In the above we used the fact that

$$C_3\,(q)\ \wedge\ 0 < L \le U \quad \implies \quad 2L + q_4(U-L) > 0.$$

Note that the function

$$f_3(\sqrt{x}) := \left(\sqrt{x}\right)^2 - ((2-q_4)L + q_4U)\sqrt{x} + (1-q_1-q_2)L^2 + q_1LU + q_2U^2,$$

is convex and its critical point

$$x_C = \frac{(2-q_4)L + q_4U}{2}$$

lies in the interval $[L, U]$. Hence, we have

$$
\begin{aligned}
& \underset{L,U,x}{\forall}\ 0 < L \ \le \sqrt{x} \le U \implies f_3\left(\sqrt{x}\right) \ge 0 \\
\iff & \underset{L,U}{\forall}\ 0 < L \le x_c \le U \implies f\left(x_C\right) \ge 0 \\
\iff & \underset{L,U}{\forall}\ 0 < L \le \frac{(2-q_4)L + q_4U}{2} \le U \\
& \implies -\left(\frac{(2-q_4)L + q_4U}{2}\right)^2 + (1-q_1-q_2)L^2 + q_1LU + q_2U^2 \ge 0.
\end{aligned}
$$

By using *QEPCAD-B*, we obtained

$$
\begin{aligned}
& \underset{L,U}{\forall}\ 0 < L \le \frac{(2-q_4)L + q_4U}{2} \le U \\
& \implies -\left(\frac{(2-q_4)L + q_4U}{2}\right)^2 + (1-q_1-q_2)L^2 + q_1LU + q_2U^2 \ge 0 \\
\iff & q_4^2 - 4q_2 \le 0\ \wedge\ q_1 + 2q_2 - q_4 \ge 0.
\end{aligned}
$$

□

By Lemma 3.21, we have $p_2 = 0$. Hence from now on, we will replace p_2 with 0.

Lemma 3.26. *$E(p,q)$ is the same as the function in the file at* `http://www.risc.jku.at/people/merascu/PhDThesis/SquareRoot/PI-PI.nb`.

Proof. We have

$$\begin{aligned} E\left(p,q\right) &= \sup_{\substack{L,U,x\\ 0<L<\sqrt{x}<U}} \left[\frac{\frac{(1-q_1-q_2)L^2+q_1LU+q_2U^2+x}{(2-q_4)L+q_4U}-\frac{(1-p_1-p_2)L^2+p_1LU+p_2U^2+x}{(2-p_4)L+p_4U}}{U-L}\right] \\ &= \sup_{\substack{L,U,x\\ 0<L<\sqrt{x}<U}} \left[\frac{\frac{(1-q_1-q_2)L^2+q_1LU+q_2U^2+x}{(2-q_4)L+q_4U}-\frac{(1-p_1)L^2+p_1LU+x}{(2-p_4)L+p_4U}}{U-L}\right]. \end{aligned}$$

By removing the denominators and collecting, we have

$$\begin{aligned} E\left(p,q\right) = \sup_{\substack{L,U,x\\ 0<L<\sqrt{x}<U}} \Bigg[& \frac{\begin{array}{c}((1-q_1-q_2)L^2+q_1LU+q_2U^2)((2-p_4)L+p_4U)\\ -((1-p_1)L^2+p_1LU)((2-q_4)L+q_4U)\end{array}}{(U-L)((2-p_4)L+p_4U)((2-q_4)L+q_4U)} \\ & +\frac{((2-p_4)L+p_4U)-((2-q_4)L+q_4U)}{(U-L)((2-p_4)L+p_4U)((2-q_4)L+q_4U)}x\Bigg] \end{aligned}$$

Note that the function

$$\begin{aligned} g(x) := & \frac{\begin{array}{c}((1-q_1-q_2)L^2+q_1LU+q_2U^2)((2-p_4)L+p_4U)\\ -((1-p_1)L^2+p_1LU)((2-q_4)L+q_4U)\end{array}}{(U-L)((2-p_4)L+p_4U)((2-q_4)L+q_4U)} \\ & +\frac{((2-p_4)L+p_4U)-((2-q_4)L+q_4U)}{(U-L)((2-p_4)L+p_4U)((2-q_4)L+q_4U)}x \end{aligned}$$

attains its maximum at

$$\begin{array}{lll} g(U^2) & \text{if} & p_4-q_4\geq 0\\ \wedge & & \\ g(L^2) & \text{if} & p_4-q_4\leq 0. \end{array}$$

In the above we used the fact that

$$\begin{aligned} & C\left(p,q\right)\ \wedge\ 0<L\leq U\ \wedge\ \frac{((2-p_4)L+p_4U)-((2-q_4)L+q_4U)}{(U-L)((2-p_4)L+p_4U)((2-q_4)L+q_4U)}\geq 0 \\ \iff\ & C\left(p,q\right)\ \wedge\ 0<L\leq U\ \wedge\ \frac{(p_4-q_4)(U-L)}{(U-L)((2-p_4)L+p_4U)((2-q_4)L+q_4U)}\geq 0 \\ \iff\ & C\left(p,q\right)\ \wedge\ 0<L\leq U\ \wedge\ \frac{p_4-q_4}{((2-p_4)L+p_4U)((2-q_4)L+q_4U)}\geq 0 \\ \iff\ & C\left(p,q\right)\ \wedge\ 0<L\leq U\ \wedge\ p_4-q_4\geq 0. \end{aligned}$$

Similarly

$$C\,(p,q)\ \wedge\ 0<L\leq U\ \wedge\ \frac{((2-p_4)L+p_4U)-((2-q_4)L+q_4U)}{(U-L)((2-p_4)L+p_4U)((2-q_4)L+q_4U)}\leq 0$$
$$\Longleftrightarrow\ p_4-q_4\leq 0.$$

Hence

$$\begin{aligned} E\,(p,q) &= \sup_{\substack{L,U,x\\ 0<L<\sqrt{x}<U}} g(x)\\ &= \sup\{E_1\,(p,q)\,,E_2\,(p,q)\}, \end{aligned}$$

where

$$\begin{aligned} E_1\,(p,q) &:= \sup_{\substack{L,U\\ 0<L<U}} g(U^2) \quad \text{if} \quad p_4-q_4\geq 0\\ E_2\,(p,q) &:= \sup_{\substack{L,U\\ 0<L<U}} g(L^2) \quad \text{if} \quad p_4-q_4\leq 0. \end{aligned}$$

Similar to Lemma 3.15, in order to compute $E(p,q)$ we proceed as follows. Note that $E_1(p,q)$ and $E_2(p,q)$ have, respectively, the form

$$\begin{aligned} E_1(p,q) &= \ v_{1i}(p,q) \quad \text{if} \quad c_{1i}(p,q), \qquad i=1..n\\ E_2(p,q) &= \ v_{2j}(p,q) \quad \text{if} \quad c_{2j}(p,q), \qquad j=1..m \end{aligned}$$

Hence, $E(p,q)$ has the form

$$E(p,q)=\begin{cases} v_{1i}(p,q) & \text{if} \quad c_{1i}(p,q)\ \wedge\ c_{2j}(p,q)\ \wedge\ v_{1i}(p,q)\geq v_{2j}(p,q)\\ v_{2j}(p,q) & \text{if} \quad c_{1i}(p,q)\ \wedge\ c_{2j}(p,q)\ \wedge\ v_{1i}(p,q)< v_{2j}(p,q) \end{cases},$$

where $i=1..n$, $j=1..m$.
Finally, we combined

$$c_{1i}(p,q)\ \wedge\ c_{2j}(p,q)\ \wedge\ v_{1i}(p,q)\geq v_{2j}(p,q)$$

and, respectively,

$$c_{1i}(p,q)\ \wedge\ c_{2j}(p,q)\ \wedge\ v_{1i}(p,q)< v_{2j}(p,q),$$

where $i=1..n$, $j=1..m$ with

$$\begin{aligned} &1\leq p_4\leq 2\ \wedge\ p_1\geq 0\ \wedge\ p_2=0\ \wedge\ p_1-p_4\leq 0\\ &\wedge\\ &0\leq q_4\leq 2\ \wedge\ q_4^2-4q_2\leq 0\ \wedge\ q_1+2q_2-q_4\geq 0. \end{aligned}$$

The quantifier-free equivalent of $E(p,q)$ is too lengthy and is not listed here, but at `http://www.risc.jku.at/people/merascu/PhDThesis/SquareRoot/PI-PI.nb`. □

Lemma 3.27. *Let* $p_4 - q_4 \geq 0$ *and*

$$E_1(p,q) := \sup_{\substack{L,U \\ 0<L<U}} g(U^2),$$

where

$$g(x) := \frac{\begin{array}{l}((1-q_1-q_2)L^2 + q_1LU + q_2U^2)((2-p_4)L + p_4U) \\ -((1-p_1)L^2 + p_1LU)((2-q_4)L + q_4U)\end{array}}{(U-L)((2-p_4)L + p_4U)((2-q_4)L + q_4U)} + \frac{((2-p_4)L + p_4U) - ((2-q_4)L + q_4U)}{(U-L)((2-p_4)L + p_4U)((2-q_4)L + q_4U)}U^2$$

The quantifier-free equivalent of $E_1(p,q)$ *is too lengthy and is not listed here, but at* `http://www.risc.jku.at/people/merascu/PhDThesis/SquareRoot/PI-PI.nb`.

Proof. Let $p_4 - q_4 \geq 0$. We have

$$\begin{aligned}
E_1(p,q) &:= \sup_{\substack{L,U \\ 0<L<U}} g(U^2) \\
&= \sup_{\substack{L,U \\ 0<L<U}} \left[\frac{\begin{array}{l}((1-q_1-q_2)L^2 + q_1LU + q_2U^2)((2-p_4)L + p_4U) \\ -((1-p_1)L^2 + p_1LU)((2-q_4)L + q_4U)\end{array}}{(U-L)((2-p_4)L + p_4U)((2-q_4)L + q_4U)} \right. \\
&\qquad \left. + \frac{((2-p_4)L + p_4U) - ((2-q_4)L + q_4U)}{(U-L)((2-p_4)L + p_4U)((2-q_4)L + q_4U)}U^2 \right] \\
&= \sup_{\substack{L,U \\ 0<L<U}} \left[\frac{\begin{array}{l}((1-q_1-q_2)L^2 + q_1LU + (q_2+1)U^2)((2-p_4)L + p_4U) \\ -((1-p_1)L^2 + p_1LU + U^2)((2-q_4)L + q_4U)\end{array}}{(U-L)((2-p_4)L + p_4U)((2-q_4)L + q_4U)} \right] \\
&= \sup_{\substack{L,U \\ 0<L<U}} \left[\frac{\begin{array}{l}(q_1L(U-L) + q_2(U-L)(U+L) + L^2 + U^2)((2-p_4)L + p_4U) \\ -(p_1L(U-L) + L^2 + U^2)((2-q_4)L + q_4U)\end{array}}{(U-L)((2-p_4)L + p_4U)((2-q_4)L + q_4U)} \right] \\
&= \sup_{\substack{L,U \\ 0<L<U}} \left[\frac{\begin{array}{l}(q_1L(U-L) + q_2(U-L)(U+L))((2-p_4)L + p_4U) \\ -(p_1L(U-L))((2-q_4)L + q_4U) \\ +(L^2 + U^2)((2-p_4)L + p_4U - ((2-q_4)L + q_4U))\end{array}}{(U-L)((2-p_4)L + p_4U)((2-q_4)L + q_4U)} \right]
\end{aligned}$$

$$= \sup_{\substack{L,U \\ 0<L<U}} \left[\frac{\begin{array}{l}(q_1 L(U-L) + q_2(U-L)(U+L))((2-p_4)L + p_4 U) \\ -(p_1 L(U-L))((2-q_4)L + q_4 U) \\ +(p_4 - q_4)(L^2+U^2)(U-L)\end{array}}{(U-L)((2-p_4)L + p_4 U)((2-q_4)L + q_4 U)} \right]$$

$$= \sup_{\substack{L,U \\ 0<L<U}} \left[\frac{\begin{array}{l}(q_1 L + q_2(U+L))((2-p_4)L + p_4 U) - p_1 L((2-q_4)L + q_4 U) \\ +(p_4 - q_4)(L^2+U^2)\end{array}}{((2-p_4)L + p_4 U)((2-q_4)L + q_4 U)} \right].$$

By dividing by $L > 0$, we have

$$E_1(p,q) = \sup_{\substack{L,U \\ 0<L<U}} \left[\frac{\begin{array}{l}(q_1 L + q_2(U+L))((2-p_4)L + p_4 U) - p_1 L((2-q_4)L + q_4 U) \\ +(p_4 - q_4)(L^2+U^2)\end{array}}{((2-p_4)L + p_4 U)((2-q_4)L + q_4 U)} \right]$$

$$= \sup_{\substack{L,U \\ 1<\frac{U}{L}}} \left[\frac{\begin{array}{l}\left(q_1 + q_2\left(\frac{U}{L}+1\right)\right)\left((2-p_4) + p_4\frac{U}{L}\right) - p_1\left((2-q_4) + q_4\frac{U}{L}\right) \\ +(p_4 - q_4)\left(1 + \left(\frac{U}{L}\right)^2\right)\end{array}}{\left((2-p_4) + p_4\frac{U}{L}\right)\left((2-q_4) + q_4\frac{U}{L}\right)} \right].$$

By renaming $\frac{U}{L}$ with K we have

$$E_1(p,q) = \sup_{\substack{K \\ K>1}} \left[\frac{(q_1 + q_2(K+1))((2-p_4) + p_4 K) - p_1((2-q_4) + q_4 K) + (p_4 - q_4)(1+K^2)}{((2-p_4) + p_4 K)((2-q_4) + q_4 K)} \right]$$

$$= \sup_{\substack{K \\ K-1>0}} \left[\frac{\begin{array}{l}(q_1 + 2q_2 + q_2(K-1))(2 + p_4(K-1)) - p_1(2 + q_4(K-1)) \\ +(p_4 - q_4)((K-1)^2 + 2(K-1) + 2)\end{array}}{(2 + p_4(K-1))(2 + q_4(K-1))} \right].$$

By renaming $K - 1$ with K we have

$$E_1(p,q) = \sup_{\substack{K \\ K>0}} \left[\frac{(q_1 + 2q_2 + q_2 K)(2 + p_4 K) - p_1(2 + q_4 K) + (p_4 - q_4)(K^2 + 2K + 2)}{(2 + p_4 K)(2 + q_4 K)} \right].$$

By performing polynomial division in the variable K, $E_1(p,q)$ can be rewritten as

$$E_1(p,q) = \frac{q_2}{q_4} - \frac{1}{p_4} + \frac{1}{q_4} + \frac{1}{p_4 q_4} \sup_{\substack{K \\ K>0}} \left[\frac{aR + b}{(2 + p_4 K)(2 + q_4 K)} \right].$$

where

$$\begin{aligned} a &= p_4^2(-2-2q_2+2q_4+q_1q_4+2q_2q_4)+q_4^2(2-2p_4-p_1p_4)\\ b &= -2p_4(2+2q_2+p_1q_4-q_1q_4-2q_2q_4+q_4^2)+2q_4(p_4^2+2). \end{aligned}$$

We computed

$$\sup_{\substack{K\\K>0}}\left[\frac{aR+b}{(2+p_4K)(2+q_4K)}\right]$$

using symbolic constrained optimization available in *Mathematica*. The quantifier-free equivalent of $E_1(p,q)$ is too lengthy and is not listed here, but at `http://www.risc.jku.at/people/merascu/PhDThesis/SquareRoot/PI-PI.nb`. ☐

Lemma 3.28. *Let $p_4-q_4\leq 0$ and*

$$E_2\,(p,q) := \sup_{\substack{L,U\\0<L<U}} g(L^2),$$

where

$$g(x) := \frac{\begin{array}{l}((1-q_1-q_2)L^2+q_1LU+q_2U^2)((2-p_4)L+p_4U)\\ -((1-p_1)L^2+p_1LU)((2-q_4)L+q_4U)\end{array}}{(U-L)((2-p_4)L+p_4U)((2-q_4)L+q_4U)} + \frac{((2-p_4)L+p_4U)-((2-q_4)L+q_4U)}{(U-L)((2-p_4)L+p_4U)((2-q_4)L+q_4U)}x$$

The quantifier-free equivalent of $E_2(p,q)$ is too lengthy and is not listed here, but at `http://www.risc.jku.at/people/merascu/PhDThesis/SquareRoot/PI-PI.nb`.

Proof. Let $p_4-q_4\leq 0$. We have

$$\begin{aligned} E_2\,(p,q) &:= \sup_{\substack{L,U\\0<L<U}} g(L^2)\\ &= \sup_{\substack{L,U\\0<L<U}}\left[\frac{\begin{array}{l}((1-q_1-q_2)L^2+q_1LU+q_2U^2)((2-p_4)L+p_4U)\\ -((1-p_1)L^2+p_1LU)((2-q_4)L+q_4U)\end{array}}{(U-L)((2-p_4)L+p_4U)((2-q_4)L+q_4U)}\right.\\ &\qquad\left.+\frac{((2-p_4)L+p_4U)-((2-q_4)L+q_4U)}{(U-L)((2-p_4)L+p_4U)((2-q_4)L+q_4U)}L^2\right]\\ &= \sup_{\substack{L,U\\0<L<U}}\left[\frac{\begin{array}{l}((2-q_1-q_2)L^2+q_1LU+q_2U^2)((2-p_4)L+p_4U)\\ -((2-p_1)L^2+p_1LU)((2-q_4)L+q_4U)\end{array}}{(U-L)((2-p_4)L+p_4U)((2-q_4)L+q_4U)}\right] \end{aligned}$$

$$= \sup_{\substack{L,U \\ 0<L<U}} \left[\frac{\begin{array}{l}(2L^2 + q_1L(U-L) + q_2(U-L)(U+L))((2-p_4)L + p_4U) \\ -(2L^2 + p_1L(U-L))((2-q_4)L + q_4U)\end{array}}{(U-L)((2-p_4)L + p_4U)((2-q_4)L + q_4U)} \right]$$

$$= \sup_{\substack{L,U \\ 0<L<U}} \left[\frac{\begin{array}{l}(q_1L(U-L) + q_2(U-L)(U+L))((2-p_4)L + p_4U) \\ -(p_1L(U-L))((2-q_4)L + q_4U) \\ +2L^2(((2-p_4)L + p_4U) - (((2-q_4)L + q_4U)))\end{array}}{(U-L)((2-p_4)L + p_4U)((2-q_4)L + q_4U)} \right]$$

$$= \sup_{\substack{L,U \\ 0<L<U}} \left[\frac{\begin{array}{l}(q_1L(U-L) + q_2(U-L)(U+L))((2-p_4)L + p_4U) \\ -(p_1L(U-L))((2-q_4)L + q_4U) \\ +2L^2(p_4 - q_4)(U-L)\end{array}}{(U-L)((2-p_4)L + p_4U)((2-q_4)L + q_4U)} \right]$$

$$= \sup_{\substack{L,U \\ 0<L<U}} \left[\frac{\begin{array}{l}(q_1L + q_2(U+L))((2-p_4)L + p_4U) - p_1L((2-q_4)L + q_4U) \\ +2L^2(p_4 - q_4)\end{array}}{((2-p_4)L + p_4U)((2-q_4)L + q_4U)} \right].$$

By dividing by $L > 0$, we have

$$E_2(p,q) = \sup_{\substack{L,U \\ 1<\frac{U}{L}}} \left[\frac{\left(q_1 + q_2\left(\frac{U}{L} + 1\right)\right)\left((2-p_4) + p_4\frac{U}{L}\right) - p_1\left((2-q_4) + q_4\frac{U}{L}\right) + 2(p_4 - q_4)}{\left((2-p_4) + p_4\frac{U}{L}\right)\left((2-q_4) + q_4\frac{U}{L}\right)} \right].$$

By renaming $\frac{U}{L}$ with K we have

$$\begin{aligned} E_2(p,q) &= \sup_{\substack{K \\ K>1}} \left[\frac{(q_1 + q_2(K+1))((2-p_4) + p_4K) - p_1((2-q_4) + q_4K) + 2(p_4 - q_4)}{((2-p_4) + p_4K)((2-q_4) + q_4K)} \right] \\ &= \sup_{\substack{K \\ K-1>0}} \left[\frac{(q_1 + 2q_2 + q_2(K-1))(2 + p_4(K-1)) - p_1(2 + q_4(K-1)) + 2(p_4 - q_4)}{(2 + p_4(K-1))(2 + q_4(K-1))} \right]. \end{aligned}$$

By renaming $K-1$ with K we have

$$E_2(p,q) = \sup_{\substack{K \\ K>0}} \left[\frac{(q_1 + 2q_2 + q_2K)(2 + p_4K) - p_1(2 + q_4K) + 2(p_4 - q_4)}{(2 + p_4K)(2 + q_4K)} \right].$$

By performing polynomial division in the variable K, $E_2(p,q)$ can be rewritten as

$$E_2(p,q) = \frac{q_2}{q_4} + \frac{1}{q_4} \sup_{\substack{K \\ K>0}} \left[\frac{cR + d}{(2 + p_4K)(2 + q_4K)} \right],$$

where

$$c = p_4(-2q_2 + q_1q_4 + 2q_2q_4) - p_1q_4^2$$
$$d = q_4(-2p_1 + 2p_4 + 2q_1 + 4q_2 - q_4) - 4q_2.$$

We computed

$$\sup_{\substack{K\\K>0}} \left[\frac{cR+d}{(2+p_4K)(2+q_4K)}\right]$$

using symbolic constrained optimization available in *Mathematica*. The quantifier-free equivalent of $E_2(p,q)$ is too lengthy and is not listed here, but at `http://www.risc.jku.at/people/merascu/PhDThesis/SquareRoot/PI-PI.nb`. $\square$

Lemma 3.29 (Main Theorem (b)). *We have*

$$E(p,q) = \frac{1}{4} \iff \begin{array}{l} p_0 = 1 - p_1 - p_2 \ \wedge\ p_3 = 2 - p_4 \\ \wedge \\ q_0 = 1 - q_1 - q_2 \ \wedge\ q_3 = 2 - q_4 \\ \wedge \\ q_1 \le p_1 \le 1 \ \wedge\ p_2 = 0 \ \wedge\ p_4 = 1 \\ \wedge \\ \frac{1}{2} \le q_1 \le 1 \ \wedge\ q_2 = \frac{1}{4} \ \wedge\ q_4 = 1 \end{array}$$

Proof. We follow the reasoning for obtaining the proof of Theorem 3.14. The computations were carried out with *Mathematica* computer algebra system. Finally, we obtained the claim.

The complete computational process for finding the constraints on p, q can be found at `http://www.risc.jku.at/people/merascu/PhDThesis/SquareRoot/PI-PI.nb`. $\square$

Lemma 3.30 (Main Theorem (a)). *We have* $E(p,q) \ge \frac{1}{4}$.

Proof. The proof is immediate, since $C(p^*,q^*)$ is true and $E(p^*,q^*) = \frac{1}{4}$, where $p^* = (0,1,0,1,1)$, $q^* = (\frac{1}{4},\frac{1}{2},\frac{1}{4},1,1)$. $\square$

It would be desirable to find the optimal refinement map satisfying $C(p,q)$, similarly to what was proved in Section 3.4. We were able to prove that R_{p^*,q^*}, where $p^* = (0,1,0,1,1)$, $q^* = (\frac{1}{4},\frac{1}{2},\frac{1}{4},1,1)$, is optimal among the quadratic maps satisfying $C'(p,q)$, but not over $C(p,q)$ (Secant-Newton map is a counterexample). Currently, we are investigating if the set of values of L, U, x for which R_{p^*,q^*} is not optimal has measure-zero.

4. Conclusion and Future Work

This thesis presents algorithmic methods for building and maintaining reliable software through *program analysis* and *synthesis*. To achieve this goal, we combine theoretical research motivated by practical applications in formal methods, automated theorem proving, and computer algebra.

Our static analysis approach shows that reasoning about imperative programs, in particular those with iterative structures, does not necessarily need a complex theoretical construction, because: *i*) it is possible to transfer the semantics of the program into the semantics of the logical formulas, thus avoiding any special theory related to program execution; *ii*) the termination condition can be expressed as an induction principle in the object theory of the domain manipulated by the program.

Currently, our method can be applied to programs with single recursion and with arbitrarily-nested loops with abrupt termination.

We also synthesized optimal algorithms for computing the square root of a real number by iterative refining. This was achieved by: *i*) transforming the synthesis problem into a program verification problem, *ii*) imposing natural assumptions in order to simplify the solution process, and *iii*) applying quantifier elimination techniques in order to solve the program verification problem.

The research performed in this thesis spans research areas ranging from program specification and verification, automated theorem proving to error analysis and interval analysis and computer algebra. Moreover, it revealed new research agenda in the direction of the development of efficient automated theorem provers, quantifier elimination and constraint optimization algorithms which we plan to investigate further as follows.

It is interesting to study how our verification method can be extended to handle other types of abrupt termination and recursion and to programs with data structures.

We also plan to research the optimality of other classes of refining functions. The final goal is to consider the full class. At this aim, we have to develop and/or adapt powerful algebraic algorithms to our specific problem. Since SMT technology owns and continuously develops efficient algorithms, and recently algebraic algorithms for quantifier elimination (cylindrical algebraic decomposition) have been adapted to this technology, we plan to explore their application and extension to our problem.

Furthermore, the ideas from the square root algorithm analysis and synthesis can, apparently, be applied for nth root computation algorithm. We also plan to consider this extension of our work.

A. Theorema Proofs. Simple Loops

A.1. Existence of the Recursion Index

Prove:

(Theorem (Existence of the recursion index))

$$\underset{\delta}{\forall}\Big(\iota[\delta] \Rightarrow \underset{n}{\exists}\Big(\neg\phi\left[R^n[\delta]\right] \wedge \underset{m}{\forall}\left(\neg\phi\left[R^m[\delta]\right] \Rightarrow m \geq n\right)\Big)\Big),$$

under the assumptions:

(Definition (Termination))

$$\underset{\delta}{\forall}(\iota[\delta] \Rightarrow (\neg\phi[\delta] \Rightarrow \pi[\delta]) \wedge (\phi[\delta] \wedge \pi[R[\delta]] \Rightarrow \pi[\delta])) \Rightarrow \underset{\delta}{\forall}(\iota[\delta] \Rightarrow \pi[\delta]),$$

(Assumption (Instantiation of π))

$$\underset{\delta}{\forall}\Big(\pi[\delta] :\Longleftrightarrow \quad \underset{n}{\exists}\Big(\neg\phi\left[R^n[\delta]\right] \wedge \underset{m}{\forall}\left(\neg\phi\left[R^m[\delta]\right] \Rightarrow m \geq n\right)\Big)\Big),$$

(Assumption (Rn+)) $\underset{n,x}{\forall}\Big(R^n[R[x]] {:=} R^{n^+}[x]\Big)$,

(Assumption (R0)) $\underset{\delta}{\forall}\big(R^0[\delta] {:=} \delta\big)$,

(Assumption (Prop. Nat. 1)) $\underset{n}{\forall}(n \geq 0)$,

(Assumption (n-+)) $\underset{n}{\forall}\Big(n \neq 0 \Rightarrow \Big(\left(n^-\right)^+ {:=} n\Big)\Big)$,

(Assumption (xy++)) $\underset{x,y}{\forall}\left(x \geq y \Rightarrow x^+ \geq y^+\right)$.

From (Definition (Termination)), by (Assumption (Instantiation of π)), we obtain:

(1)
$$\begin{aligned}
&\underset{\delta}{\forall}\Big(\iota[\delta] \Rightarrow \Big(\neg\phi[\delta] \Rightarrow \underset{n}{\exists}\Big(\neg\phi\left[R^n[\delta]\right] \wedge \underset{m}{\forall}\left(\neg\phi\left[R^m[\delta]\right] \Rightarrow m \geq n\right)\Big)\Big) \wedge \\
&\qquad\Big(\quad \phi[\delta] \wedge \underset{n}{\exists}\Big(\neg\phi\left[R^n[R[\delta]]\right] \wedge \underset{m}{\forall}\left(\neg\phi\left[R^m[R[\delta]]\right] \Rightarrow m \geq n\right)\Big) \Rightarrow \\
&\qquad\qquad \underset{n}{\exists}\Big(\neg\phi\left[R^n[\delta]\right] \wedge \underset{m}{\forall}\left(\neg\phi\left[R^m[\delta]\right] \Rightarrow m \geq n\right)\Big)\Big)\Big) \\
&\Rightarrow \\
&\underset{\delta}{\forall}\Big(\iota[\delta] \Rightarrow \underset{n}{\exists}\Big(\neg\phi\left[R^n[\delta]\right] \wedge \underset{m}{\forall}\left(\neg\phi\left[R^m[\delta]\right] \Rightarrow m \geq n\right)\Big)\Big).
\end{aligned}$$

From (1), by (Assumption (Rn+)), we obtain:

$$(2)\quad \underset{\delta}{\forall}\Big(\iota[\delta] \Rightarrow \Big(\neg\phi[\delta] \Rightarrow \underset{n}{\exists}\Big(\neg\phi\,[R^n[\delta]] \wedge \underset{m}{\forall}(\neg\phi\,[R^m[\delta]] \Rightarrow m \geq n)\Big)\Big) \wedge \Big(\ \phi[\delta] \wedge \underset{n}{\exists}\Big(\neg\phi\left[R^{n^+}[\delta]\right] \wedge \underset{m}{\forall}\Big(\neg\phi\left[R^{m^+}[\delta]\right] \Rightarrow m \geq n\Big)\Big) \Rightarrow \underset{n}{\exists}\Big(\neg\phi\,[R^n[\delta]] \wedge \underset{m}{\forall}(\neg\phi\,[R^m[\delta]] \Rightarrow m \geq n)\Big)\Big)\Big) \Rightarrow \underset{\delta}{\forall}\Big(\iota[\delta] \Rightarrow \underset{n}{\exists}\Big(\neg\phi\,[R^n[\delta]] \wedge \underset{m}{\forall}(\neg\phi\,[R^m[\delta]] \Rightarrow m \geq n)\Big)\Big).$$

For proving (Theorem (Existence of the recursion index)), by (2), it suffices to prove

$$(4)\quad \underset{\delta}{\forall}\Big(\iota[\delta] \Rightarrow \Big(\neg\phi[\delta] \Rightarrow \underset{n}{\exists}\Big(\neg\phi\,[R^n[\delta]] \wedge \underset{m}{\forall}(\neg\phi\,[R^m[\delta]] \Rightarrow m \geq n)\Big)\Big) \wedge \Big(\ \phi[\delta] \wedge \underset{n}{\exists}\Big(\neg\phi\left[R^{n^+}[\delta]\right] \wedge \underset{m}{\forall}\Big(\neg\phi\left[R^{m^+}[\delta]\right] \Rightarrow m \geq n\Big)\Big) \Rightarrow \underset{n}{\exists}\Big(\neg\phi\,[R^n[\delta]] \wedge \underset{m}{\forall}(\neg\phi\,[R^m[\delta]] \Rightarrow m \geq n)\Big)\Big)\Big).$$

For proving (4) we take all variables arbitrary but fixed and prove:

$$(5)\quad \iota\,[\delta_0] \Rightarrow \Big(\neg\phi\,[\delta_0] \Rightarrow \underset{n}{\exists}\Big(\neg\phi\,[R^n\,[\delta_0]] \wedge \underset{m}{\forall}(\neg\phi\,[R^m\,[\delta_0]] \Rightarrow m \geq n)\Big)\Big) \Big(\ \phi\,[\delta_0] \wedge \underset{n}{\exists}\Big(\neg\phi\left[R^{n^+}\,[\delta_0]\right] \wedge \underset{m}{\forall}\Big(\neg\phi\left[R^{m^+}\,[\delta_0]\right] \Rightarrow m \geq n\Big)\Big) \Rightarrow \underset{n}{\exists}\Big(\neg\phi\,[R^n\,[\delta_0]] \wedge \underset{m}{\forall}(\neg\phi\,[R^m\,[\delta_0]] \Rightarrow m \geq n)\Big)\Big).$$

We prove (5) by the deduction rule.

We assume

$$(6)\quad \iota\,[\delta_0]$$

and show

$$(7)\quad \Big(\neg\phi\,[\delta_0] \Rightarrow \underset{n}{\exists}\Big(\neg\phi\,[R^n\,[\delta_0]] \wedge \underset{m}{\forall}(\neg\phi\,[R^m\,[\delta_0]] \Rightarrow m \geq n)\Big)\Big) \wedge \Big(\ \phi\,[\delta_0] \wedge \underset{n}{\exists}\Big(\neg\phi\left[R^{n^+}\,[\delta_0]\right] \wedge \underset{m}{\forall}\Big(\neg\phi\left[R^{m^+}\,[\delta_0]\right] \Rightarrow m \geq n\Big)\Big) \Rightarrow \underset{n}{\exists}\Big(\neg\phi\,[R^n\,[\delta_0]] \wedge \underset{m}{\forall}(\neg\phi\,[R^m\,[\delta_0]] \Rightarrow m \geq n)\Big)\Big).$$

To prove (7) one has to prove

$$(8)\quad \neg\phi\,[\delta_0] \Rightarrow \neg\phi\left[R^0\,[\delta_0]\right] \wedge \underset{m}{\forall}(\neg\phi\,[R^m\,[\delta_0]] \Rightarrow m \geq 0) \text{ and assumes}$$

$$(9)\quad \phi\,[\delta_0] \wedge \Big(\neg\phi\left[R^{{n_0}^+}\,[\delta_0]\right] \wedge \underset{m}{\forall}\Big(\neg\phi\left[R^{m^+}\,[\delta_0]\right] \Rightarrow m \geq n_0\Big)\Big) \text{ and proves}$$

$$(10)\quad \neg\phi\left[R^{{n_0}^+}\,[\delta_0]\right] \wedge \underset{m}{\forall}(\neg\phi\,[R^m\,[\delta_0]] \Rightarrow m \geq {n_0}^+).$$

We prove (8) by the deduction rule.

We assume

$$(11)\quad \neg\phi\,[\delta_0]$$

and show

$$(12)\quad \neg\phi\left[R^0\,[\delta_0]\right] \wedge \underset{m}{\forall}(\neg\phi\,[R^m\,[\delta_0]] \Rightarrow m \geq 0).$$

We prove the individual conjunctive parts of (12):

Proof of (12.1) $\quad \neg\phi\left[R^0\left[\delta_0\right]\right]$:
Using (Assumption (R0)), the goal (12.1) is transformed into:
(13) $\quad \neg\phi\left[\delta_0\right]$.
Formula (13) is true because it is identical to (11).
Proof of (12.2) $\quad \underset{m}{\forall}\left(\neg\phi\left[R^m\left[\delta_0\right]\right] \Rightarrow m \geq 0\right)$:
For proving (12.2) we take all variables arbitrary but fixed and prove:
(15) $\quad \neg\phi\left[R^{m_0}\left[\delta_0\right]\right] \Rightarrow m_0 \geq 0$.
We prove (15) by the deduction rule.
We assume
(16) $\quad \neg\phi\left[R^{m_0}\left[\delta_0\right]\right]$
and show
(17) $\quad m_0 \geq 0$.
From (17), by (Assumption (Prop. Nat. 1)), we obtain:
(18) $\quad m_0 \geq 0$.
Formula (17) is true because it is identical to (18).
We prove the individual conjunctive parts of (10):
Proof of (10.1) $\quad \neg\phi\left[R^{n_0{}^+}\left[\delta_0\right]\right]$:
Formula (10.1) is true because it is identical to (9.2.1).
Proof of (10.2) $\quad \underset{m}{\forall}\left(\neg\phi\left[R^m\left[\delta_0\right]\right] \Rightarrow m \geq n_0{}^+\right)$:
For proving (10.2) we take all variables arbitrary but fixed and prove:
(19) $\quad \neg\phi\left[R^{m_1}\left[\delta_0\right]\right] \Rightarrow m_1 \geq n_0{}^+$.
We prove (19) by the deduction rule.
We assume
(20) $\quad \neg\phi\left[R^{m_1}\left[\delta_0\right]\right]$
and show
(21) $\quad m_1 \geq n_0{}^+$.
From (20), and (9.1), we obtain:
(22) $\quad m_1 \neq 0$.
From (22), by (Assumption (n-+)), we obtain:
(24) $\quad (m_1{}^-)^+ := m_1$.
From (22) and an appropriate instance of (9.2.2) we obtain by modus ponens:
(25) $\quad \neg\phi\left[R^{(m_1{}^-)^+}\left[\delta_0\right]\right] \Rightarrow m_1{}^- \geq n_0$.
From (25), by (24), we obtain:
(26) $\quad \neg\phi\left[R^{m_1}\left[\delta_0\right]\right] \Rightarrow m_1{}^- \geq n_0$.
From (20) and (26) we obtain by modus ponens
(27) $\quad m_1{}^- \geq n_0$.

From (27), by (Assumption (xy++)), we obtain:
(28) $({m_1}^-)^+ \geq {n_0}^+$.
From (28), by (24), we obtain:
(29) $m_1 \geq {n_0}^+$.
Formula (21) is true because it is identical to (29). □

A.2. Existence of the Function Implemented by the Loop

Prove:
(Theorem (Semantics)) $\underset{\delta}{\forall}(\iota[\delta] \Rightarrow (\neg\phi[\delta] \Rightarrow (f[\delta] = \delta)) \wedge (\phi[\delta] \Rightarrow (f[\delta] = f[R[\delta]])))$,
under the assumptions:
(Definition (Witness)) $\underset{\delta}{\forall}\left(\iota[\delta] \Rightarrow \left(f[\delta]{:=}R^{M[\delta]}[\delta]\right)\right)$,
(Assumption (M[δ])) $\underset{\delta}{\forall}(\iota[\delta] \Rightarrow (\neg\phi[\delta] \Rightarrow (M[\delta]{:=}0)))$,
(Assumption (R0)) $\underset{x}{\forall}\left(R^0[x]{:=}x\right)$,
(Assumption (Rn+)) $\underset{n,x}{\forall}\left(R^n[R[x]]{:=}R^{n^+}[x]\right)$,
(Assumption (R+)) $\underset{\delta}{\forall}(\iota[\delta] \Rightarrow (M[R[\delta]]^+{:=}M[\delta]))$,
(Definition (Loop Safety)) $\underset{\delta}{\forall}(\iota[\delta] \Rightarrow (\phi[\delta] \Rightarrow \iota[R[\delta]]))$.
For proving (Theorem (Semantics)) we take all variables arbitrary but fixed and prove:
(1) $\iota[\delta_0] \Rightarrow (\neg\phi[\delta_0] \Rightarrow (f[\delta_0] = \delta_0)) \wedge (\phi[\delta_0] \Rightarrow (f[\delta_0] = f[R[\delta_0]]))$.
We prove (1) by the deduction rule.
We assume
(2) $\iota[\delta_0]$
and show
(3) $(\neg\phi[\delta_0] \Rightarrow (f[\delta_0] = \delta_0)) \wedge (\phi[\delta_0] \Rightarrow (f[\delta_0] = f[R[\delta_0]]))$.
We prove the individual conjunctive parts of (3):
Proof of (3.1) $\neg\phi[\delta_0] \Rightarrow (f[\delta_0] = \delta_0)$:
We prove (3.1) by the deduction rule.
We assume
(4) $\neg\phi[\delta_0]$
and show
(5) $f[\delta_0] = \delta_0$.
From (2), by (Assumption (M[δ])), we obtain:
(8) $\neg\phi[\delta_0] \Rightarrow (M[\delta_0]{:=}0)$.
From (2), by (Definition (Witness)), we obtain:
(7) $f[\delta_0]{:=}R^{M[\delta_0]}[\delta_0]$.

From (4) and (8) we obtain by modus ponens

(11) $M[\delta_0] := 0$.

Using (7), the goal (5) is transformed into:

(12) $R^{M[\delta_0]}[\delta_0] = \delta_0$.

Using (11), the goal (12) is transformed into:

(13) $R^0[\delta_0] = \delta_0$.

Using (Assumption (R0)), the goal (13) is transformed into:

(14) $\delta_0 = \delta_0$.

Formula (14) is proved because is True.

Proof of (3.2) $\quad \phi[\delta_0] \Rightarrow (f[\delta_0] = f[R[\delta_0]])$:

We prove (3.2) by the deduction rule.

We assume

(15) $\phi[\delta_0]$

and show

(16) $f[\delta_0] = f[R[\delta_0]]$.

From (2), by (Definition (Loop Safety)), we obtain:

(21) $\phi[\delta_0] \Rightarrow \iota[R[\delta_0]]$.

From (2), by (Assumption (R+)), we obtain:

(20) $M[R[\delta_0]]^+ := M[\delta_0]$.

From (2), by (Definition (Witness)), we obtain:

(18) $f[\delta_0] := R^{M[\delta_0]}[\delta_0]$.

From (15) and (21) we obtain by modus ponens

(22) $\iota[R[\delta_0]]$.

Using (18), the goal (16) is transformed into:

(23) $R^{M[\delta_0]}[\delta_0] = f[R[\delta_0]]$.

From (22), by (Definition (Witness)), we obtain:

(24) $f[R[\delta_0]] := R^{M[R[\delta_0]]}[R[\delta_0]]$.

Using (24), the goal (23) is transformed into:

(28) $R^{M[\delta_0]}[\delta_0] = R^{M[R[\delta_0]]}[R[\delta_0]]$.

Using (Assumption (Rn+)), the goal (28) is transformed into:

(29) $R^{M[\delta_0]}[\delta_0] = R^{M[R[\delta_0]]^+}[\delta_0]$.

Using (20), the goal (29) is transformed into:

(30) $R^{M[\delta_0]}[\delta_0] = R^{M[\delta_0]}[\delta_0]$.

Formula (30) is proved because is True. $\square$

A.3. Uniqueness of the Function Implemented by the Loop

Prove:

(Theorem (Uniqueness of f)) $\underset{\delta}{\forall}(\iota[\delta] \Rightarrow (f[\delta] = g[\delta]))$,

under the assumptions:

(Definition (Termination))

$$\underset{\delta}{\forall}(\iota[\delta] \Rightarrow (\neg\phi[\delta] \Rightarrow \pi[\delta]) \wedge (\phi[\delta] \wedge \pi[R[\delta]] \Rightarrow \pi[\delta])) \Rightarrow \underset{\delta}{\forall}(\iota[\delta] \Rightarrow \pi[\delta]),$$

(Definition (Semantics of f))

$$\underset{\delta}{\forall}(\iota[\delta] \Rightarrow (\neg\phi[\delta] \Rightarrow (f[\delta]{:=}\delta)) \wedge (\phi[\delta] \Rightarrow (f[\delta]{:=}f[R[\delta]]))),$$

(Definition (Semantics of g))

$$\underset{\delta}{\forall}(\iota[\delta] \Rightarrow (\neg\phi[\delta] \Rightarrow (g[\delta]{:=}\delta)) \wedge (\phi[\delta] \Rightarrow (g[\delta]{:=}g[R[\delta]]))),$$

(Assumption (Instantiation of π)) $\underset{\delta}{\forall}(\pi[\delta] :\Leftrightarrow\ f[\delta] = g[\delta])$.

From (Definition (Termination)), by (Assumption (Instantiation of π)), we obtain:

(1) $\underset{\delta}{\forall}(\iota[\delta] \Rightarrow (\neg\phi[\delta] \Rightarrow (f[\delta] = g[\delta])) \wedge (\phi[\delta] \wedge (f[R[\delta]] = g[R[\delta]]) \Rightarrow (f[\delta] = g[\delta])))$
$\Rightarrow$
$\underset{\delta}{\forall}(\iota[\delta] \Rightarrow (f[\delta] = g[\delta]))$.

For proving (Theorem (Uniqueness of f)), by (1), it suffices to prove

(3) $\underset{\delta}{\forall}(\iota[\delta] \Rightarrow (\neg\phi[\delta] \Rightarrow (f[\delta] = g[\delta])) \wedge (\phi[\delta] \wedge (f[R[\delta]] = g[R[\delta]]) \Rightarrow (f[\delta] = g[\delta])))$.

For proving (3) we take all variables arbitrary but fixed and prove:

(4) $\iota[\delta_0] \Rightarrow$
$(\neg\phi[\delta_0] \Rightarrow (f[\delta_0] = g[\delta_0])) \wedge (\phi[\delta_0] \wedge (f[R[\delta_0]] = g[R[\delta_0]]) \Rightarrow (f[\delta_0] = g[\delta_0]))$.

We prove (4) by the deduction rule.

We assume

(5) $\iota[\delta_0]$

and show

(6) $(\neg\phi[\delta_0] \Rightarrow (f[\delta_0] = g[\delta_0])) \wedge (\phi[\delta_0] \wedge (f[R[\delta_0]] = g[R[\delta_0]]) \Rightarrow (f[\delta_0] = g[\delta_0]))$.

We prove the individual conjunctive parts of (6):

Proof of (6.1) $\neg\phi[\delta_0] \Rightarrow (f[\delta_0] = g[\delta_0])$:

We prove (6.1) by the deduction rule.

We assume

(7) $\neg\phi[\delta_0]$

and show

(8) $f[\delta_0] = g[\delta_0]$.
From (7), by (Definition (Semantics of g)), we obtain:
(10) $\iota[\delta_0] \Rightarrow (g[\delta_0] := \delta_0) \wedge (\phi[\delta_0] \Rightarrow (g[\delta_0] := g[R[\delta_0]]))$.
From (7), by (Definition (Semantics of f)), we obtain:
(9) $\iota[\delta_0] \Rightarrow (f[\delta_0] := \delta_0) \wedge (\phi[\delta_0] \Rightarrow (f[\delta_0] := f[R[\delta_0]]))$.
From (5) and (9) we obtain by modus ponens
(13) $(f[\delta_0] := \delta_0) \wedge (\phi[\delta_0] \Rightarrow (f[\delta_0] := f[R[\delta_0]]))$.
From (5) and (10) we obtain by modus ponens
(14) $(g[\delta_0] := \delta_0) \wedge (\phi[\delta_0] \Rightarrow (g[\delta_0] := g[R[\delta_0]]))$.
Using (13.1), the goal (8) is transformed into:
(15) $\delta_0 = g[\delta_0]$.
(14.1), the goal (15) is transformed into:
(16) $\delta_0 = \delta_0$.
Formula (16) is proved because is True.
Proof of (6.2) $\phi[\delta_0] \wedge (f[R[\delta_0]] = g[R[\delta_0]]) \Rightarrow (f[\delta_0] = g[\delta_0])$:
We prove (6.2) by the deduction rule.
We assume
(17) $\phi[\delta_0] \wedge (f[R[\delta_0]] = g[R[\delta_0]])$
and show
(18) $f[\delta_0] = g[\delta_0]$.
From (17.1), by (Definition (Semantics of g)), we obtain:
(20) $\iota[\delta_0] \Rightarrow (\phi[\delta_0] \Rightarrow (g[\delta_0] := g[R[\delta_0]]))$.
From (17.1), by (Definition (Semantics of f)), we obtain:
(19) $\iota[\delta_0] \Rightarrow (\phi[\delta_0] \Rightarrow (f[\delta_0] := f[R[\delta_0]]))$.
From (5) and (19) we obtain by modus ponens
(23) $\phi[\delta_0] \Rightarrow (f[\delta_0] := f[R[\delta_0]])$.
From (17.1) and (23) we obtain by modus ponens
(24) $f[\delta_0] := f[R[\delta_0]]$.
From (5) and (20) we obtain by modus ponens
(25) $\phi[\delta_0] \Rightarrow (g[\delta_0] := g[R[\delta_0]])$.
From (17.1) and (25) we obtain by modus ponens
(26) $g[\delta_0] := g[R[\delta_0]]$.
Using (24), the goal (18) is transformed into:
(27) $f[R[\delta_0]] = g[\delta_0]$.
Using (26), the goal (27) is transformed into:
(28) $f[R[\delta_0]] = g[R[\delta_0]]$.
Formula (28) is true because it is identical to (17.2). $\square$

A.4. Total Correctness

Prove:

(Theorem (Total Correctness))

$$\underset{\delta}{\forall}(\iota[\delta] \Rightarrow \iota[f[\delta]]),$$

under the assumptions:

(Definition (Termination))

$$\underset{\delta}{\forall}(\iota[\delta] \Rightarrow (\neg\phi[\delta] \Rightarrow \pi[\delta]) \wedge (\phi[\delta] \wedge \pi[R[\delta]] \Rightarrow \pi[\delta])) \Rightarrow \underset{\delta}{\forall}(\iota[\delta] \Rightarrow \pi[\delta]),$$

(Definition (Semantics))

$$\underset{\delta}{\forall}(\iota[\delta] \Rightarrow (\neg\phi[\delta] \Rightarrow (f[\delta]{:=}\delta)) \wedge (\phi[\delta] \Rightarrow (f[\delta]{:=}f[R[\delta]]))),$$

(Assumption (Instantiation of π))

$$\underset{\delta}{\forall}(\pi[\delta] :\Longleftrightarrow \quad \iota[f[\delta]]).$$

From (Definition (Termination)), by (Assumption (Instantiation of π)), we obtain:

(1) $\underset{\delta}{\forall}(\iota[\delta] \Rightarrow (\neg\phi[\delta] \Rightarrow \iota[f[\delta]]) \wedge (\phi[\delta] \wedge \iota[f[R[\delta]]] \Rightarrow \iota[f[\delta]])) \Rightarrow \underset{\delta}{\forall}(\iota[\delta] \Rightarrow \iota[f[\delta]])$.

For proving (Theorem (Total Correctness)), by (1), it suffices to prove

(3) $\underset{\delta}{\forall}(\iota[\delta] \Rightarrow (\neg\phi[\delta] \Rightarrow \iota[f[\delta]]) \wedge (\phi[\delta] \wedge \iota[f[R[\delta]]] \Rightarrow \iota[f[\delta]]))$.

For proving (3) we take all variables arbitrary but fixed and prove:

(4) $\iota[\delta_0] \Rightarrow (\neg\phi[\delta_0] \Rightarrow \iota[f[\delta_0]]) \wedge (\phi[\delta_0] \wedge \iota[f[R[\delta_0]]] \Rightarrow \iota[f[\delta_0]])$.

We prove (4) by the deduction rule.

We assume

(5) $\iota[\delta_0]$

and show

(6) $(\neg\phi[\delta_0] \Rightarrow \iota[f[\delta_0]]) \wedge (\phi[\delta_0] \wedge \iota[f[R[\delta_0]]] \Rightarrow \iota[f[\delta_0]])$.

From (5), by (Definition (Semantics)), we obtain:

(7) $(\neg\phi[\delta_0] \Rightarrow (f[\delta_0]{:=}\delta_0)) \wedge (\phi[\delta_0] \Rightarrow (f[\delta_0]{:=}f[R[\delta_0]]))$.

We prove the individual conjunctive parts of (6):

Proof of (6.1) $\quad \neg\phi[\delta_0] \Rightarrow \iota[f[\delta_0]]$:

We prove (6.1) by the deduction rule.

We assume

(8) $\neg\phi[\delta_0]$

and show

(9) $\iota\,[f\,[\delta_0]]$.

From (8) and (7.1) we obtain by modus ponens

(10) $f\,[\delta_0] := \delta_0$.

Using (10), the goal (9) is transformed into:

(11) $\iota\,[\delta_0]$.

Formula (11) is true because it is identical to (5).

Proof of (6.2) $\quad \phi\,[\delta_0] \wedge \iota\,[f\,[R\,[\delta_0]]] \Rightarrow \iota\,[f\,[\delta_0]]$:

We prove (6.2) by the deduction rule.

We assume

(12) $\phi\,[\delta_0] \wedge \iota\,[f\,[R\,[\delta_0]]]$

and show

(13) $\iota\,[f\,[\delta_0]]$.

From (12.1) and (7.2) we obtain by modus ponens

(14) $f\,[\delta_0] := f\,[R\,[\delta_0]]$.

Using (14), the goal (13) is transformed into:

(15) $\iota\,[f\,[R\,[\delta_0]]]$.

Formula (15) is true because it is identical to (12.2). $\square$

B. Theorema Proofs. Loops with `return`

B.1. Total Correctness

Prove:

(Theorem (Total Correctness))

$$\underset{\alpha,\delta}{\forall}\left(I_P[\alpha] \wedge \iota[\delta] \Rightarrow (\neg\phi[\delta] \Rightarrow \iota[f[\delta]]) \wedge (\phi[\delta] \wedge \psi[\delta] \Rightarrow O_P[\alpha, S[\delta]])\right),$$

under the assumptions:

(Definition (Semantics))

$$\underset{\delta}{\forall}(\iota[\delta] \Rightarrow (\neg\phi[\delta] \Rightarrow (f[\delta]{:=}\delta)) \wedge (\phi[\delta] \wedge \psi[\delta] \Rightarrow (f[\delta]{:=}S[\delta]))),$$

(Definition (Functional Correctness)) $\underset{\alpha,\delta}{\forall}(I_P[\alpha] \wedge \iota[\delta] \Rightarrow (\phi[\delta] \wedge \psi[\delta] \Rightarrow O_P[\alpha, S[\delta]]))$.

For proving (Theorem (Total Correctness)) we take all variables arbitrary but fixed and prove:

(1) $I_P[\alpha_0] \wedge \iota[\delta_0] \Rightarrow (\neg\phi[\delta_0] \Rightarrow \iota[f[\delta_0]]) \wedge (\phi[\delta_0] \wedge \psi[\delta_0] \Rightarrow O_P[\alpha_0, S[\delta_0]])$.

We prove (1) by the deduction rule.

We assume

(2) $I_P[\alpha_0] \wedge \iota[\delta_0]$

and show

(3) $(\neg\phi[\delta_0] \Rightarrow \iota[f[\delta_0]]) \wedge (\phi[\delta_0] \wedge \psi[\delta_0] \Rightarrow O_P[\alpha_0, S[\delta_0]])$.

From (2), by (Definition (Functional Correctness)), we obtain:

(4) $\phi[\delta_0] \wedge \psi[\delta_0] \Rightarrow O_P[\alpha_0, S[\delta_0]]$.

From (2.2), by (Definition (Semantics)), we obtain:

(5) $(\neg\phi[\delta_0] \Rightarrow (f[\delta_0]{:=}\delta_0)) \wedge (\phi[\delta_0] \wedge \psi[\delta_0] \Rightarrow (f[\delta_0]{:=}S[\delta_0]))$.

We prove the individual conjunctive parts of (3):

Proof of (3.1) $\neg\phi[\delta_0] \Rightarrow \iota[f[\delta_0]]$:

We prove (3.1) by the deduction rule.

We assume

(6) $\neg\phi[\delta_0]$

and show

(7) $\iota[f[\delta_0]]$.

From (6) and (5.1) we obtain by modus ponens
(8) $f[\delta_0] := \delta_0$.
Using (8), the goal (7) is transformed into:
(9) $\iota[\delta_0]$.
Formula (9) is true because it is identical to (2.2).
Proof of (3.2) $\phi[\delta_0] \wedge \psi[\delta_0] \Rightarrow O_P[\alpha_0, S[\delta_0]]$:
Formula (3.2) is true because it is identical to (4). $\square$

C. Mathematica Routines and Listings Accompanying Section 3.5

C.1. The function $E(p,q)$ from Lemma 3.15

$$E(p,q) = \begin{cases} E_1 & \text{if} \quad G_1 \\ E_2 & \text{if} \quad G_2 \\ E_3 & \text{if} \quad G_3 \\ E_4 & \text{if} \quad G_4 \end{cases}$$

where

$$\begin{aligned} E_1 &= 1 - \frac{1}{p_4} \\ E_2 &= 1 - \frac{1}{q_4} \\ E_3 &= 1 - \frac{2}{p_3 + p_4} \\ E_4 &= 1 - \frac{2}{q_3 + q_4} \end{aligned} \qquad \begin{aligned} G_1 &= g_{11} \ \vee \ \dots \ \vee \ g_{16} \\ G_2 &= g_{21} \ \vee \ \dots \ \vee \ g_{25} \\ G_3 &= g_{31} \ \vee \ \dots \ \vee \ g_{35} \\ G_4 &= g_{41} \ \vee \ \dots \ \vee \ g_{43} \end{aligned}$$

where again

$$\begin{aligned} g_{11} =& 1 - \frac{1}{p_4} \geq 1 - \frac{2}{q_3 + q_4} \ \wedge \ p_4 - q_4 > 0 \ \wedge \ -p_3 - p_4 + q_3 + q_4 > 0 \\ & \wedge \ p_3 - p_4 \leq 0 \ \wedge \ p_3 + p_4 - q_3 - q_4 \leq 0 \ \wedge \ q_3 - q_4 > 0 \ \wedge \ q_4 \geq 2 \\ & \wedge \ q_3 \geq 2 - q_4 \ \wedge \ p_4 \geq 1 \ \wedge \ p_3 \geq 2 - p_4 \\ g_{12} =& 1 - \frac{1}{p_4} \geq 1 + \frac{-p_3 + p_4 + q_3 - q_4}{-p_4 q_3 + p_3 q_4} \ \wedge \ p_4 - q_4 > 0 \ \wedge \ -p_3 - p_4 + q_3 + q_4 > 0 \\ & \wedge \ p_3 - p_4 \leq 0 \ \wedge \ -p_4 + q_4 < 0 \ \wedge \ p_3 + p_4 - q_3 - q_4 \leq 0 \\ & \wedge \ q_3 - q_4 \leq 0 \ \wedge \ q_4 \geq 2 \ \wedge \ q_3 \geq 2 - q_4 \wedge p_4 \geq 1 \ \wedge \ p_3 \geq 2 - p_4 \end{aligned}$$

$$
\begin{aligned}
g_{13} =&1-\frac{1}{p_4}\geq 1-\frac{2}{q_3+q_4} \ \wedge \ p_4-q_4\geq 0 \ \wedge \ -p_3-p_4+q_3+q_4\leq 0\\
&\wedge \ p_3-p_4\leq 0 \ \wedge \ p_3+p_4-q_3-q_4\leq 0 \ \wedge \ q_3-q_4>0 \ \wedge \ q_4\geq 2\\
&\wedge \ q_3\geq 2-q_4 \ \wedge \ p_4\geq 1 \ \wedge \ p_3\geq 2-p_4\\
g_{14} =&1-\frac{1}{p_4}\geq 1-\frac{1}{q_4} \ \wedge \ p_4-q_4\geq 0 \ \wedge \ -p_3-p_4+q_3+q_4\leq 0\\
&\wedge \ p_3-p_4\leq 0 \ \wedge \ -p_4+q_4\geq 0 \ \wedge \ p_3+p_4-q_3-q_4\leq 0 \ \wedge \ q_3-q_4\leq 0\\
&\wedge \ q_4\geq 2 \ \wedge \ q_3\geq 2-q_4 \ \wedge \ p_4\geq 1 \ \wedge \ p_3\geq 2-p_4\\
g_{15} =&p_4-q_4\geq 0 \ \wedge \ -p_3-p_4+q_3+q_4\leq 0 \wedge \ p_3-p_4\leq 0 \ \wedge \ -p_4+q_4\leq 0\\
&\wedge \ p_3+p_4-q_3-q_4>0 \ \wedge \ q_4\geq 2 \ \wedge \ q_3\geq 2-q_4 \ \wedge \ p_4\geq 1 \ \wedge \ p_3\geq 2-p_4\\
g_{16} =&1-\frac{1}{p_4}\geq 1+\frac{-p_3+p_4+q_3-q_4}{-p_4q_3+p_3q_4} \ \wedge \ p_4-q_4\geq 0 \ \wedge \ -p_3-p_4+q_3+q_4\leq 0\\
&\wedge \ p_3-p_4\leq 0 \ \wedge \ -p_4+q_4<0 \ \wedge \ p_3+p_4-q_3-q_4\leq 0 \ \wedge \ q_3-q_4\leq 0\\
&\wedge \ q_4\geq 2 \ \wedge \ q_3\geq 2-q_4 \wedge p_4\geq 1 \ \wedge \ p_3\geq 2-p_4
\end{aligned}
$$

$$
\begin{aligned}
g_{21} =&1-\frac{2}{p_3+p_4}<1-\frac{1}{q_4} \ \wedge \ -p_3-p_4+q_3+q_4\leq 0 \ \wedge \ p_3-p_4>0\\
&\wedge \ -p_4+q_4>0 \ \wedge \ p_3+p_4-q_3-q_4>0 \ \wedge \ q_3-q_4\leq 0 \ \wedge \ q_4\geq 2\\
&\wedge \ q_3\geq 2-q_4 \ \wedge \ p_4\geq 1 \ \wedge \ p_3\geq 2-p_4\\
g_{22} =&1-\frac{2}{p_3+p_4}<1-\frac{1}{q_4} \ \wedge \ -p_3-p_4+q_3+q_4\leq 0 \ \wedge \ p_3-p_4>0\\
&\wedge \ -p_4+q_4\geq 0 \ \wedge \ p_3+p_4-q_3-q_4\leq 0 \ \wedge \ q_3-q_4\leq 0 \ \wedge \ q_4\geq 2\\
&\wedge \ q_3\geq 2-q_4 \ \wedge \ p_4\geq 1 \ \wedge \ p_3\geq 2-p_4\\
g_{23} =&p_4-q_4\leq 0 \ \wedge \ -p_3-p_4+q_3+q_4>0 \ \wedge \ -p_4+q_4\geq 0 \ \wedge \ p_3+p_4-q_3-q_4\leq 0\\
&\wedge \ q_3-q_4\leq 0 \ \wedge \ q_4\geq 2 \ \wedge \ q_3\geq 2-q_4 \ \wedge \ p_4\geq 1 \ \wedge \ p_3\geq 2-p_4\\
g_{24} =&1+\frac{p_3-p_4-q_3+q_4}{p_4q_3-p_3q_4}<1-\frac{1}{q_4} \ \wedge \ p_4-q_4<0 \ \wedge \ -p_3-p_4+q_3+q_4\leq 0\\
&\wedge \ p_3-p_4\leq 0 \ \wedge \ -p_4+q_4>0 \ \wedge \ p_3+p_4-q_3-q_4>0 \ \wedge \ q_3-q_4\leq 0\\
&\wedge \ q_4\geq 2 \ \wedge \ q_3\geq 2-q_4 \wedge p_4\geq 1 \ \wedge \ p_3\geq 2-p_4\\
g_{25} =&1+\frac{p_3-p_4-q_3+q_4}{p_4q_3-p_3q_4}<1-\frac{1}{q_4} \ \wedge \ p_4-q_4<0 \ \wedge \ -p_3-p_4+q_3+q_4\leq 0\\
&\wedge \ p_3-p_4\leq 0 \ \wedge \ -p_4+q_4\geq 0 \ \wedge \ p_3+p_4-q_3-q_4\leq 0 \ \wedge \ q_3-q_4\leq 0\\
&\wedge \ q_4\geq 2 \ \wedge \ q_3\geq 2-q_4 \wedge p_4\geq 1 \ \wedge \ p_3\geq 2-p_4
\end{aligned}
$$

$$
\begin{aligned}
g_{31} =& 1 - \frac{2}{p_3 + p_4} \geq 1 + \frac{-p_3 + p_4 + q_3 - q_4}{-p_4 q_3 + p_3 q_4} \ \wedge\ -p_3 - p_4 + q_3 + q_4 \leq 0 \ \wedge\ p_3 - p_4 > 0 \\
& \wedge\ -p_4 + q_4 > 0 \ \wedge\ p_3 + p_4 - q_3 - q_4 > 0 \ \wedge\ q_3 - q_4 > 0 \ \wedge\ q_4 \geq 2 \ \wedge\ q_3 \geq 2 - q_4 \\
& \wedge\ p_4 \geq 1 \ \wedge\ p_3 \geq 2 - p_4
\end{aligned}
$$

$$
\begin{aligned}
g_{32} =& 1 - \frac{2}{p_3 + p_4} \geq 1 - \frac{1}{q_4} \ \wedge\ -p_3 - p_4 + q_3 + q_4 \leq 0 \ \wedge\ p_3 - p_4 > 0 \ \wedge\ -p_4 + q_4 > 0 \\
& \wedge\ p_3 + p_4 - q_3 - q_4 > 0 \ \wedge\ q_3 - q_4 \leq 0 \ \wedge\ q_4 \geq 2 \ \wedge\ q_3 \geq 2 - q_4 \\
& \wedge\ p_4 \geq 1 \ \wedge\ p_3 \geq 2 - p_4 \\
g_{33} =& 1 - \frac{2}{p_3 + p_4} \geq 1 - \frac{2}{q_3 + q_4} \ \wedge\ -p_3 - p_4 + q_3 + q_4 \leq 0 \ \wedge\ p_3 - p_4 > 0 \\
& \wedge\ p_3 + p_4 - q_3 - q_4 \leq 0 \ \wedge\ q_3 - q_4 > 0 \ \wedge\ q_4 \geq 2 \ \wedge\ q_3 \geq 2 - q_4 \\
& \wedge\ p_4 \geq 1 \ \wedge\ p_3 \geq 2 - p_4 \\
g_{34} =& 1 - \frac{2}{p_3 + p_4} \geq 1 - \frac{1}{q_4} \ \wedge\ -p_3 - p_4 + q_3 + q_4 \leq 0 \ \wedge\ p_3 - p_4 > 0 \ \wedge\ -p_4 + q_4 \geq 0 \\
& \wedge\ p_3 + p_4 - q_3 - q_4 \leq 0 \ \wedge\ q_3 - q_4 \leq 0 \ \wedge\ q_4 \geq 2 \ \wedge\ q_3 \geq 2 - q_4 \\
& \wedge\ p_4 \geq 1 \ \wedge\ p_3 \geq 2 - p_4 \\
g_{35} =& - p_3 - p_4 + q_3 + q_4 \leq 0 \ \wedge\ p_3 - p_4 > 0 \ \wedge\ -p_4 + q_4 \leq 0 \ \wedge\ \ p_3 + p_4 - q_3 - q_4 > 0 \\
& \wedge\ q_4 \geq 2 \ \wedge\ q_3 \geq 2 - q_4 \ \wedge\ p_4 \geq 1 \ \wedge\ p_3 \geq 2 - p_4
\end{aligned}
$$

$$
\begin{aligned}
g_{41} =& 1 + \frac{p_3 - p_4 - q_3 + q_4}{p_4 q_3 - p_3 q_4} < 1 - \frac{2}{q_3 + q_4} \ \wedge\ p_4 - q_4 > 0 \ \wedge\ -p_3 - p_4 + q_3 + q_4 > 0 \\
& \wedge\ p_3 - p_4 > 0 \ \wedge\ p_3 + p_4 - q_3 - q_4 \leq 0 \ \wedge\ q_3 - q_4 > 0 \ \wedge\ q_4 \geq 2 \ \wedge\ q_3 \geq 2 - q_4 \\
& \wedge\ p_4 \geq 1 \ \wedge\ p_3 \geq 2 - p_4 \\
g_{42} =& 1 - \frac{1}{p_4} < 1 - \frac{2}{q_3 + q_4} \ \wedge\ p_4 - q_4 > 0 \ \wedge\ -p_3 - p_4 + q_3 + q_4 > 0 \\
& \wedge\ p_3 - p_4 \leq 0 \ \wedge\ p_3 + p_4 - q_3 - q_4 \leq 0 \ \wedge\ q_3 - q_4 > 0 \ \wedge\ q_4 \geq 2 \ \wedge\ q_3 \geq 2 - q_4 \\
& \wedge\ p_4 \geq 1 \ \wedge\ p_3 \geq 2 - p_4 \\
g_{43} =& p_4 - q_4 \leq 0 \ \wedge\ -p_3 - p_4 + q_3 + q_4 > 0 \ \wedge\ p_3 + p_4 - q_3 - q_4 \leq 0 \\
& \wedge\ q_3 - q_4 > 0 \ \wedge\ q_4 \geq 2 \ \wedge\ q_3 \geq 2 - q_4 \ \wedge\ p_4 \geq 1 \ \wedge\ p_3 \geq 2 - p_4
\end{aligned}
$$

C.2. Constrained Optimization Routine for Proof of Theorem Theorem 3.14

The routine `FindMin` has the following specification.

- Input: list of the form $\{\{e_1, c_1\}, ..., \{e_n, c_n\}\}$, where e_i is an expression in p, q, c_i is a conjunction of equalities/inequalities in p, q, $i = 1..n$;

- Output: list of the form $\{\{\{v_1, s_1\}, C_1\}, ..., \{\{v_n, s_n\}, C_n\}\}$, where v_i is the minimum value in the region determined by c_i, s_i is a substitution for p, q for which $e_i = v_i$, C_i is a disjunction of conjunctions of equalities/inequalities in p, q for which $e_i = v_i$, $i = 1..n$.

```
Clear[FindMin];
FindMin[l_] := Module[{min, minLst={}},
    For[i=1, i<=Length[l], i++,
    min = Minimize[l[[i]], {p3, p4, q3, q4}];
    R = LogicalExpand[Reduce[
                        Join[{min[[1]]>=l[[i]][[1]]}, l[[i]][[2;;-1]]],
Reals]];
    minLst = Append[minLst, {min,R}]; ]; minLst];
```

C.3. Output of the Routine `FindMin`

```
{{{1/2, {p3 -> 2, p4 -> 2, q3 -> 2, q4 -> 2}}, False},
 {{1/2, {p3 -> 0, p4 -> 2, q3 -> 2, q4 -> 2}}, False},
 {{1/2, {p3 -> 0, p4 -> 2, q3 -> 2, q4 -> 2}}, False},
 {{1/2, {p3 -> 0, p4 -> 2, q3 -> 0, q4 -> 2}}, False},
 {{1/2, {p3 -> 3, p4 -> 1, q3 -> 2, q4 -> 2}}, False},
 {{1/2, {p3 -> 5/2, p4 -> 3/2, q3 -> 1, q4 -> 2}},
  p4==4-p3 && q4==2 && 2<p3 && q3<-2+p3+p4 && 0<=q3 && p3<=3},
 {{1/2, {p3 -> 2, p4 -> 3/2, q3 -> 3/4, q4 -> 2}},
  (q4==2 && 1<p3 && p4<p3 && q3<-2+p3+p4 && 0<=q3 && 1<=p4 && p3<=2) ||
  (q4==2 && 2<p3 && p3<3  && p4<4-p3 && q3<-2+p3+p4 && 0<=q3 && 1<=p4)},
 {{1/2, {p3 -> 3, p4 -> 1, q3 -> 2, q4 -> 2}}, False},
 {{1/2, {p3 -> 5/2, p4 -> 3/2, q3 -> 2, q4 -> 2}},
  4-p4==p3 && q3==2 && q4==2 && p4<2 && 1<=p4},
 {{1/2, {p3 -> 7/4, p4 -> 5/4, q3 -> 1, q4 -> 2}},
  2-p4+q3==p3 && q4==2 && p4<2 && -2+2p4<q3 && q3<2 && 1<=p4},
 {{1/2, {p3 -> 2, p4 -> 2, q3 -> 2, q4 -> 2}}, False},
 {{1/2, {p3 -> 2, p4 -> 2, q3 -> 2, q4 -> 2}}, False},
 {{1/2, {p3 -> 2, p4 -> 0, q3 -> 0, q4 -> 2}},
  p4==2 && q3==p3 && q4==2 && 0<=q3 && q3<=2},
 {{1/2, {p3 -> 2, p4 -> 2, q3 -> 0, q4 -> 2}},
  p4==2 && q4==2 && 0<p3 && q3<p3 && 0<=q3 && p3<=2},
 {{1/2, {p3 -> 0, p4 -> 2, q3 -> 0, q4 -> 2}}, False},
 {{1/2, {p3 -> 2, p4 -> 2, q3 -> 2, q4 -> 2}}, False},
 {{1/2, {p3 -> 3, p4 -> 1, q3 -> 2, q4 -> 2}},
  (p4==2-p3 && q4==2 && 0<q3 && 0<=p3 && p3<=1 && q3<=2) ||
```

```
 (q4==2 && 1<p3 && p3<2 && -2+p3+p4<q3 && 1<=p4 && p4<=2 && q3<=2) ||
 (q4==2 && 2-p3<p4 && -2+p3+p4<q3 && 0<=p3 && p3<=1 && p4<=2 && q3<=2)||
 (q4==2 && p3<3 && p4<4-p3 && -2+p3+p4<q3 && 1<=p4 && 2<=p3 && q3<=2)},
{{1/2, {p3 -> 1, p4 -> 3/2, q3 -> 1/4, q4 -> 2}},
 (q4==2 && 0<p3 && 2-p3<p4 && p4<2 && q3<-2+p3+p4 && 0<=q3 && p3<=1) ||
 (q4==2 && 1<p3 && p3<2 && p4<2 && q3<-2+p3+p4 && 0<=q3 && p3<=p4)},
{{1/2, {p3 -> 5/4, p4 -> 7/4, q3 -> 1, q4 -> 2}},
 (p3==1 && p4==1 && q3==0 && q4==2) ||
 (2-p4+q3==p3 && q4==2 && 1<p4 && p4<2 && 0<=q3 && q3<=-2+2p4)}}
```

Bibliography

[1] G. Alefeld and J. Herzberger. *Introduction to Interval Computations.* Academic Press, Inc., New York, NY, 1983.

[2] S. Anand, C. Păsăreanu, and W. Visser. JPF-SE: A Symbolic Execution Extension to Java PathFinder. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2007.

[3] D. Arnon, G. E. Collins, and S. McCallum. Cylindrical Algebraic Decomposition I: The Basic Algorithm. *SIAM Journal on Computing*, 1984.

[4] B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* Springer, 2007.

[5] N. Beebe. Accurate Square Root Computation. Technical report, Center for Scientific Computing, Department of Mathematics,University of Utah, 1991.

[6] J. Berg and B. Jacobs. The LOOP Compiler for Java and JML. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2001.

[7] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Springer, 2004.

[8] R. Boute. Calculational Semantics: Deriving Programming Theories from Equations by Functional Predicate Calculus. *ACM Transactions on Programming Languages and Systems*, 2006.

[9] A. Bradley and Z. Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification.* Springer-Verlag New York, Inc., 2007.

[10] C. Brown. Simple CAD Construction and its Applications. *Journal of Symbolic Computation*, 2001.

[11] C. Brown. QEPCAD-B: A Program for Computing with Semi-Algebraic Sets using CADs. *SIGSAM Bulletin*, 2003.

[12] B. Buchberger and A. Craciun. Algorithm Synthesis by Lazy Thinking: Examples and Implementation in Theorema. In *Electronic Notes in Theoretical Computer Science*, 2004.

[13] B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, and D. Vasaru. A Survey of the Theorema Project. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, 1997.

[14] E. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Workshop on Logic of Programs*, 1982.

[15] W. Cody and W. Waite. *Software Manual for the Elementary Functions*. Prentice-Hall, Englewood Cliffs, NJ, 1980.

[16] P. J. Cohen. Decision Procedures for Real and p-adic Fields. *Communications on Pure and Applied Mathematics*, 1969.

[17] G. E. Collins. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. In *Automata Theory and Formal Languages*, 1975.

[18] B. Cook, A. Podelski, and A. Rybalchenko. Termination Proofs for Systems Code. *ACM SIGPLAN Notices*, 2006.

[19] D. Coward. Symbolic Execution Systems - a Review. *Journal of Software Engeneering*, 1988.

[20] J. Davenport and J. Heintz. Real Quantifier Elimination is Doubly Exponential. *Journal of Symbolic Computation*, 1988.

[21] L. de Moura and N. Björner. Z3: An Efficient SMT Solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[22] Department of Commerce, National Institute of Standards and Technology. Software Errors Cost U.S. Economy 59.5 Billion Annually. `http://www.nist.gov/public_affairs/releases/n02-10.htm`, 2002.

[23] E. W. Dijkstra. A Constructive Approach to the Problem of Program Correctness. *BIT*, 1968.

[24] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[25] A. Dolzmann and T. Sturm. REDLOG: Computer Algebra Meets Computer Logic. *SIGSAM Bulletin*, 1997.

[26] A. Dolzmann, T. Sturm, and V. Weispfenning. A New Approach for Automatic Theorem Proving in Real Geometry. *Journal of Automated Reasoning*, 1998.

[27] B. Buchberger et al. A Survey of the Theorema project. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, 1997.

[28] B. Buchberger et al. The Theorema Project: A Progress Report. In *Proceedings of the Symposium on the Integration of Symbolic Computation and Mechanized Reasoning*, 2000.

[29] B. Buchberger et al. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, 2006.

[30] M. J. Fischer and M. O. Rabin. Super-Exponential Complexity of Presburger Arithmetic. In *Proceedings of Complexity of Computation*, 1974.

[31] C. Flanagan, R. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended Static Checking for Java. *ACM SIGPLAN Notices*, 2002.

[32] R. Floyd. Assigning Meaning to Programs. In *Proceedings of Symposia in Symposia in Applied Mathematics American Mathematical Society*, 1967.

[33] D. Fowler and E. Robson. Square Root Approximations in Old Babylonian Math-

ematics: YBC 7289 in Context. *Historia Mathematica*, 1998.

[34] M. Gordon and T. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[35] D. Gries. *The Science of Programming*. Springer, 1981.

[36] E. R. Hansen and R. I. Greenberg. An Interval Newton Method. *Applied Mathematics and Computation*, 1983.

[37] J.F. Hart, E.W. Cheney, C.L. Lawson, H.J. Maehly, C.K. Mesztenyi, J.R. Rice, H.C. Thacher Jr., and C. Witzgall. *Computer Approximations*. John Wiley, 1968. Reprinted, E. Krieger Publishing Company (1978).

[38] J. Heintz, M. Roy, and P. Solernó. Sur la Complexité du Principe de Tarski-Seidenberg. *Bulletin de la Société Mathématique de France*, 1990.

[39] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 1969.

[40] H. Hong. An Improvement of the Projection Operator in Cylindrical Algebraic Decomposition. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, 1990.

[41] H. Hong. Simple Solution Formula Construction in Cylindrical Algebraic Decomposition based Quantifier Elimination. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, 1992.

[42] H. Hong. Quantifier Elimination for Formulas Constrained by Quadratic Equations. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, 1993.

[43] H. Hong. An Efficient Method for Analyzing the Topology of Plane Real Algebraic Curves. *Mathematics and Computers in Simulation*, 1996.

[44] W. Howden. Methodology for the Authomatic Generation of Program Test Data. Technical report, Standford Artificial Inteligence Laboratory, Standford, CA, 1973.

[45] M. Huisman. *Reasoning about Java Programs in Higher Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001.

[46] B. Jacobs, J. Smans, and F. Piessens. VeriFast: Imperative Programs as Proofs. In *Verified Software: Theories, Tools and Experiments; Workshop on Tools & Experiments*, 2010.

[47] P. Janičić and A. Bundy. Automatic Synthesis of Decision Procedures: A Case Study of Ground and Linear Arithmetic. In *Towards Mechanized Mathematical Assistants*, 2007.

[48] D. Kapur. Automatically Generating Loop Invariants using Quantifier Elimination. In *Proceedings of ACA*, 2004.

[49] D. Kapur. A Quantifier Elimination based Heuristic for Automatically Generating Inductive Assertions for Programs. *Journal of Systems Science and Complexity*,

2006.

[50] M. Kaufmann, J. Strother Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.

[51] J. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 1976.

[52] M. Kirchner. Program Verification with the Mathematical Software System Theorema. Technical report, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, 1999.

[53] L. Kovacs. *Automated Invariant Generation by Algebraic Techniques for Imperative Program Verification in Theorema.* PhD thesis, RISC, Johannes Kepler University Linz, Austria, 2007.

[54] A. Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic.* PhD thesis, Technische Universität München, 2009.

[55] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Professional, 2002.

[56] G. Leavens and Y. Cheon. Design by Contract with JML. 2003.

[57] C. S. Lee, N. Jones, and A. Ben-Amram. The Size-Change Principle for Program Termination. *ACM SIGPLAN Notices*, 2001.

[58] J. Loeckx, K. Sieber, and R. Stansifer. *The Foundations of Program Verification.* John Wiley & Sons, Inc., 1984.

[59] S. McCallum. An Improved Projection Operation for Cylindrical Algebraic Decomposition. In *Research Contributions from the European Conference on Computer Algebra-Volume 2*, 1985.

[60] S. McCallum. On Projection in CAD-based Quantifier Elimination with Equational Constraint. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, 1999.

[61] J. McCarthy. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems*, 1963.

[62] J. R. Meggitt. Pseudo Division and Pseudo Multiplication Processes. *IBM Journal of Research and Development*, 1962.

[63] J. Meyer and A. Poetzsch-Heffter. An Architecture for Interactive Program Provers. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2000.

[64] R. E. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, NJ, 1966.

[65] D. R. Morrison. A Method for Computing the Inverse of Certain Functions. *MTAC*, 1956.

[66] A. Neumaier. *Interval Methods for Systems of Equations.* Cambridge University Press, Cambridge, UK, 1990.

[67] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for*

Higher-Order Logic. Springer, 2002.

[68] M. Norrish. *C Formalized in HOL*. PhD thesis, Cambridge University, 1998.

[69] O. Olsson and A. Wallenburg. Customised Induction Rules for Proving Correctness of Imperative Programs. In *Proceedings of the International Conference on Software Engineering and Formal Methods*, 2005.

[70] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *Proceedings of the International Conference on Automated Deduction*, 1992.

[71] A. Platzer and J. Quesel. KeYmaera: A Hybrid Theorem Prover for Hybrid Systems. In *Proceedings of International Joint Conference on Automated Reasoning*, 2008.

[72] A. Poetzsch-Heffter and P. Müller. A Programming Logic for Sequential Java. In *Proceedings of European Symposium on Programming*, 1999.

[73] M. J. Cloud R. E. Moore, R. B. Kearfott. *Introduction to Interval Analysis.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 2009.

[74] J. Renegar. On the Computational Complexity and Geometry of the First-Order Theory of the Reals, Part I - III. *Journal of Symbolic Computation*, 1992.

[75] S. Sankaranaryanan, B. S. Henry, and Z. Manna. Non-linear Loop Invariant Generation using Gröbner Bases. In *Proceedings of the Symposium on Principles of Programming Languages*, 2004.

[76] W. Schreiner. Understanding Programs. Technical report, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, 2008.

[77] M. Schwerhoff. Symbolic Execution for Chalice. Master's thesis, ETH Zurich, 2011.

[78] A. Seidenberg. A New Decision Method for Elementary Algebra. *The Annals of Mathematics*, 1954.

[79] K. Slind. Function Definition in Higher-Order Logic. In *Proceedings of Theorem Proving in Higher Order Logics*, 1996.

[80] S. Srivastava, S. Gulwani, and J. Foster. From Program Verification to Program Synthesis. In *Proceedings of the Symposium on Principles of Programming Languages*, 2010.

[81] S. Steinberg and R. Liska. Stability Analysis by Quantifier Elimination. *Mathematics and Computers in Simulation*, 2002.

[82] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1981.

[83] A. Strzebonski. Cylindrical Algebraic Decomposition using Validated Numerics. *Journal of Symbolic Computation*, 2006.

[84] A. Tarski. A decision method for elementary algebra and geometry. *Bulletin of the American Mathematical Society*, 1951.

[85] A. Tiwari, N. Shankar, and J. Rushby. Invisible Formal Methods for Embed-

ded Control Systems. *Proceedings of the Institute of Electrical and Electronics Engineers*, 2003.

[86] D. von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.

[87] V. Weispfenning. The Complexity of Linear Problems in Fields. *Journal of Symbolic Computation*, 1988.

[88] V. Weispfenning. Quantifier Elimination for Real Algebra - the Cubic Case. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, 1994.

[89] V. Weispfenning. A New Approach to Quantifier Elimination for Real Algebra. In *Quantifier Elimination and Cylindrical Algebraic Decomposition, Texts and Monographs in Symbolic Computation*. Springer-Verlag, Vienna, 1998.

[90] J. H. Wensley. A Class of Non-Analytical Iterative Processes. *The Computer Journal*, 1959.

[91] F. Winkler. *Polynomial Algorithms in Computer Algebra*. Springer, 1996.

[92] S. Wolfram. *The Mathematica Book. Version 5.0.* Wolfram Media, 2003.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, daß ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfaßt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Dissertation ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, 7. November 2012 Mădălina Eraşcu

Mădălina Eraşcu

Research Institute for Symbolic Computation
Johannes Kepler University
Altenberger Strasse 69
A-4040 Linz, Austria

Phone: +43 680 4470895
Email: merascu@risc.jku.at
Homepage: http://www.risc.jku.at/home/merascu

Personal Data

Date of birth: October 10, 1983 (Oravita, Romania)

Citizenship: Romanian

Languages: Romanian (native speaker), English (fluently), Spanish (intermediate), German (intermediate), French (beginner)

Education

M.Sc. in Computer Science, Johannes Kepler University, Linz, Austria, 2008

Master's thesis: "Automated Formal Static Analysis and Retrieval of Source Code"

B.Sc. in Computer Science, West University, Timisoara, Romania, 2006

Bachelor's thesis: "XML Web Services using ADO.NET"

Current Position

Ph.D. Student in Computer Science at Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria

Ph.D. thesis: "Computational Logic and Quantifier Elimination Techniques for (Semi-)automatic Static Analysis and Synthesis of Algorithms"

Fields of Research Interest

Automated theorem proving, computer algebra, formal methods in software development, programming

Research

Journal Papers

M. Erascu and H. Hong, *Secant-Newton map is the optimal among contracting quadratic maps for square root computation*, accepted for publication in the Journal of Reliable Computing.

Refereed Conference Papers

M. Erascu and T. Jebelean, *Automated Certification of a Logic-Based Verification Method for Imperative Loops*, In Proceedings of the 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, V. Negru, A. Voronkov (ed.), to appear

M. Erascu and T. Jebelean, *A Purely Logical Approach to the Termination of Imperative Loops*, In Proceedings of the 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, S. Watt, V. Negru, T. Ida, T. Jebelean, D. Petcu, D. Zaharie (ed.), pp. 142 – 149, 2010, IEEE Computer Society, 978-0-7695-4324-6

M. Erascu and T. Jebelean, *A Calculus for Imperative Programs: Formalization and Implementation*, In Proceedings of the 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, S. Watt, V. Negru, T. Ida, T. Jebelean, D. Petcu, D. Zaharie (ed.), pp. 77 – 84, 2009, IEEE Computer Society, 978-0-7695-2964-5

Other Publications

M. Erascu and T. Jebelean, *A Purely Logical Approach to Program Termination*, In Proceedings of the 11th International Workshop on Termination, Peter Schneider-Kamp (ed.), Proceedings of Federated Logic Conference, Edinburgh, July 9–21, 2010

M. Erascu and T. Jebelean, *Practical Program Verification by Forward Symbolic Execution: Correctness and Examples*, In Austrian-Japan Workshop on Symbolic Computation in Software Science, Bruno Buchberger, Tetsuo Ida, Temur Kutsia (ed.) 08-08, pp. 47–56. 2008. RISC Report Series, University of Linz, Austria

M. Erascu, *Automated Formal Static Analysis and Retrieval of Source Code*, International School for Informatics – Johannes Kepler University. Diploma Thesis. August 2008. In RISC Raport Series 08-21

Full list: `https://www.risc.jku.at/home/merascu?view=2`

Scientific Talks

M. Erascu and H. Hong, *Semi-automatic Algorithm Analysis and Synthesis (Case Study: Square Root)*, Contributed talk at International Seminar on Program Verification, Automated Debugging and Symbolic Computation, October 10–12, 2012, Beijing, China

M. Erascu and T. Jebelean, *Automated Certification of a Logic-Based Verification Method for Imperative Loops*, Contributed talk at International Seminar on Program Verification, Automated Debugging and Symbolic Computation, October 10–12, 2012, Beijing, China

M. Erascu and H. Hong, *Semi-automatic Algorithm Analysis and Synthesis (Case Study: Square Root)*, Computer Laboratory, University of Cambridge, June 21, 2012

M. Erascu and T. Jebelean, *Automated Certification of a Logic-Based Verification Method for Imperative Loops*, Contributed talk at CiE 2012 – How the World Computes, June 18 – 23, 2012

M. Erascu, *Symbolic Computation in Static Program Analysis. Applications to Numerical Algorithms*, Contributed talk at Doctoral Program "Computational Mathematics" Statusseminar, October 5 – 7, 2011

Full list: `http://www.risc.jku.at/publications/index.php?author=merascu&division=talks`

Research Visits and Seminar Participation

Research visit at Computer Laboratory, University of Cambridge, Cambridge, UK, June 21, 2012. Host: Dr. Grant Olney Passmore

Research stay at Department of Mathematics, North Carolina State University, Raleigh, USA, March – April 2012, January – May 2011. Host: Prof. Dr. Hoon Hong

Summer School on Verification Technology, Systems & Applications, Nancy, France, 2009

Summer School on Logics and Languages for Reliability and Security, Marktoberdorf, Germany, 2009

Professional Experience

Teaching

Automated Theorem Proving

Nesin Summer School, Sirince, Turkey, July 30 – August 5, 2012

Tutorial on Cylindrical Algebraic Decomposition

Nesin Summer School, Sirince, Turkey, August 29 – September 4, 2011

Programming

Mindbreeze Software GmbH, Linz, Austria

Master's thesis, October 2007 – August 2008

S.C. memIQ S.R.L., Timisoara, Romania

Bachelor's thesis, October 2005 – May 2006

Summer internship, May 2005 – September 2005

Computer Skills

Programming languages: C, C++, Java, C#

Computer Algebra Systems: Mathematica

Honors, Awards, & Fellowships

DOC-fFORTE-fellowship of the Austrian Academy of Sciences for carrying out research at Research Institute for Symbolic Computation, June 2011 – November 2012. Competitive fellowship: 25 selections out of approx. 90 applications. (45.000 EUR)

Marshall Plan Foundation scholarship for research visit at North Carolina State University, USA, January 2011 – June 2011. (8.000 EUR)

Microsoft Research grant for the participation fee and living expenses at the Marktoberdorf Summer School, August 2009

Upper Austrian Government Scholarship for 1st year of PhD studies, 2008 – 2009

Excellency diploma awarded by the Romanian Board of Education and Research, Minister Ecaterina Andronescu, June 2002

Service

Reviewer for SCSS 2012, J. of Symbolic Computation (Special Issue Workshop on Invariant Generation 2010)

Assistant Editor for `http://www.scholarpedia.org/`, 2010 – 2011;

In charge with logistics at FLOC 2010, FPSAC 2009, SCCS 2008, RTA 2008, Calculemus 2007, WING 2007

References

Prof. Dr. Tudor Jebelean
Research Institute for Symbolic Computation
Johannes Kepler University
A-4040 Linz, Austria, Europe
email: `tjebelea@risc.jku.at`

Prof. Dr. Hoon Hong
Department of Mathematics
North Carolina State University
Box 8205, Raleigh NC 27695, USA
email: `hong@ncsu.edu`

Prof. Dr. Wolfgang Schreiner
Research Institute for Symbolic Computation
Johannes Kepler University
A-4040 Linz, Austria, Europe
email: `Wolfgang.Schreiner@risc.jku.at`

Prof. Dr. Ali Nesin
Istanbul Bilgi Üniversitesi
Matematik Bölümü
Kurtulus, Deresi Cad. 47, Dolapdere 34435 Beyoğlu Istanbul
email: `anesin@bilgi.edu.tr`

Last updated: November 21, 2012