Automated Invariant Generation by Algebraic Techniques for Imperative Program Verification in Theorema

Johannes-Kepler University Linz
Research Institute for Symbolic Computation

Automated Invariant Generation by Algebraic Techniques for Imperative Program Verification in Theorema

Laura Kovács

Doctoral Thesis

advised by

A.Univ.-Prof. Dr. Tudor Jebelean
Univ.-Prof. Dr. Andrei Voronkov

Defended in October 2007 in Linz, Austria.

The author was supported in parts by
- the Upper Austrian Government (2003/09–2004/08),
- the Austrian Ministry of Education, Science and Culture (BMBWK), the Austrian Ministry of Economy and Work (BMWA) and by the Romanian Ministry of Education and Research (MEC), in the frame of the e-Austria Project (2004/09–2005/12),
- the Austrian Science Foundation (FWF) grant SFB F 1302 (2006/01–2007/10),
- the Japan Society for the Promotion of Science (JSPS), JSPS/RCI - 2/07085 (2007/09).

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.

Linz, September 2007.

. . . . . .
Laura Kovács

Zusammenfassung

In dieser Arbeit werden algebraische und kombinatorische Zugänge zum automatischen Schließen über imperative Schleifen mit Zuweisungen, Reihungen und Bedingungen vorgestellt.

Wir definieren eine bestimmte Klasse von Schleifen, genannt „P-solvable", für die der Wert jeder Programmvariable polynomiell von Anfangswerten, dem Schleifenzähler und einigen zusätzlichen Variablen abhängt. Für solche Schleifen entwickeln wir eine systematische Methode zum Erzeugen polynomieller Gleichungen als Schleifeninvariaten.

Unter zusätztlichen Voraussetzungen ist die Methode vollständig, falls der Rumpf solcher Schleifen nur Zuweisungen und Bedingungen enthält. Das heisst, sie generiert eine Menge polynomieller Gleichungen als Invarianten, aus denen, mithilfe weiterer Annahmen für Schleifen mit Bedingungen, jede andere polynomielle Gleichung, die als Invariante dienen kann, abgeleitet werden kann. Für nicht-triviale Algorithmen, die auf Zahlen arbeiten, läßt sich zeigen, daß sie sich in der Tat mit P-solvable Schleifen implementieren lassen.

Durch Kombination fortgeschrittener Verfahren aus der algorithmischen Kombinatorik, der Symbolischen Summation, der Computer Algebra und der automatisierten Logik wird ein System zur Erzeugung polynomieller Schleifeninvarianten für imperative Programme realisiert, die auf Zahlen arbeiten.

Mithilfe der symbolischen Möglichkeiten des Computeralgebrasystems *Mathematica* werden diese Techniken in einem neuen Softwarepaket names *Aligator* implementiert. *Aligator* enthält Routinen zum Lösen verschiedener spezieller Arten von Rekurrenzen (Gosper-summierbare oder C-finite) und zum Bestimmen algebraischer Abhängigkeiten zwischen exponentiellen Folgen mit algebraischen Basen, die auf verschiedene am RISC entwickelte kombinatorische Pakete zurückgreifen. Mit *Aligator* kann für viele Algorithmen auf Zahlen erfolgreich eine vollständige Menge polynomieller Invariaten gefunden werden.

Die automatisch erhaltenen Invariaten werden anschließend zum Beweisen der partieller Korrektheit von Programmen verwendet, indem geeignete Beweisverpflichtungen als logische Formeln erster Ordnung erzeugt werden. Basierend auf Hoare-Logik und der „Weakest precondition strategy" wird dieser Verifikationsprozess von einer Verifikationsumgebung für imperative Programme im *Theorema* unterstützt. *Theorema* scheint für eine solche Integration geeignet, da es auf dem Computeralgebrasystem *Mathematica* aufsetzt und automatsche Methoden zum Beweisen in Prädikatenlogik, domain-spezifische Beweiser und Induktionsbeweiser beinhaltet.

Schlüssenwörter

Hoare Logik, Programmverifikation, Schleifeninvarianten, Symbolische Summation, Polynomielle Abhängigkeiten, Gröbnerbasen.

Abstract

This thesis presents algebraic and combinatorial approaches for reasoning about imperative loops with assignments, sequencing and conditionals.

A certain family of loops, called *P-solvable*, is defined for which the value of each program variable can be expressed as a polynomial of the initial values of variables, the loop counter, and some new variables where there are algebraic dependencies among the new variables. For such loops, a systematic method is developed for generating polynomial invariants. Further, if the bodies of these loops consist only of assignments and conditional branches, and test conditions in the loop and conditionals are ignored, the method is shown to be complete for some special cases. By completeness we mean that it generates a set of polynomials from which, under additional assumptions for loops with conditional branches, any polynomial invariant can be derived. Many non-trivial algorithms working on numbers can be naturally implemented using P-solvable loops.

By combining advanced techniques from algorithmic combinatorics, symbolic summation, computer algebra and computational logic, a framework is developed for generating polynomial invariants for imperative programs operating on numbers.

Exploiting the symbolic manipulation capabilities of the computer algebra system *Mathematica*, these techniques are implemented in a new software package called *Aligator*. By using several combinatorial packages developed at RISC, *Aligator* includes algorithms for solving special classes of recurrence relations (those that are either Gosper-summable or C-finite) and generating polynomial dependencies among algebraic exponential sequences. Using *Aligator*, a complete set of polynomial invariants is successfully generated for numerous imperative programs working on numbers.

The automatically obtained invariant assertions are subsequently used for proving the partial correctness of programs by generating appropriate verification conditions as first-order logical formulas. Based on Hoare logic and the weakest precondition strategy, this verification process is supported in an imperative verification environment implemented in the *Theorema* system. *Theorema* is convenient for such an integration given that it is built on top of the computer algebra system *Mathematica* and includes automated methods for theorem proving in predicate logic, domain specific reasoning and proving by induction.

Keywords

Hoare Logic, program verification, loop invariants, symbolic summation, polynomial relations, Gröbner bases.

Acknowledgements

Foremost, I wish to thank Bruno Buchberger and my thesis advisors, Tudor Jebelean and Andrei Voronkov.

Bruno initiated the e-Austria project on program verification between RISC and my home university. Due to his efforts to maintain and strengthen the connection between RISC and West University of Timişoara, I was offered an opportunity to complete my master studies at RISC. Moreover, his creative research interest in verification allowed me to begin my PhD studies at RISC. I thank Bruno for all his guidance, inspiration and advices during my studies. I was very lucky to be a member of the Automated Reasoning research group lead by Bruno and Tudor.

Tudor was also the advisor of my Master's thesis and I greatly appreciate his offer to continue our joint work in the framework of the RISC PhD program under his supervision. His scientific guidance and assistance, valuable suggestions, as well as the offered freedom in my research had a major role in starting, continuing and finalizing my PhD thesis. Moreover, his perfect handling of administrative issues helped me many times during my studies. Thank you once again!

I was very lucky to meet Andrei Voronkov last year, and I wish to thank him for all his support and encouragements since then. His comments, understanding of research, interest in my work and in joint research, helped me very much in moving further, towards finalizing this thesis. His remarks and suggestions, especially at the later stage of the work on this thesis, improved my research and the thesis in every aspect, and I am very grateful for all his help as an advisor and as a researcher.

Further, I wish to thank Deepak Kapur for his help and support. Although my work with Deepak started only in July 2006, his scientific remarks, ideas and comments made it possible to extend the experimental aspects of my work by theoretical issues, thus yielding a better presentation and understanding of the work I have done during my work on this thesis.

I also wish to thank the RISC community for offering a pleasant working environment, and I thank all my current and former colleagues and friends from RISC and RICAM. I especially would like to thank Wolfgang Windsteiger for his help as an advisor during my first year of PhD. Further, I wish to thank Peter Paule, whose lectures on algorithmic and analytic combinatorics gave me a better understanding of several combinatorial methods that I have been using and continue to use in my research. I would also like to thank Carsten Schneider, the first person who explained me symbolic summation. And I especially thank Manuel Kauers for all the interesting and challenging discussions related to my work, as well as for his unlimited help with LaTeX. Moreover, I wish to thank Manuel for his friendship!

I thank my professors, Viorel Negru and Dana Petcu, from my home university, West University Timişoara, for insisting on continuing my studies at RISC and being present whenever I needed their help.

I also wish to thank all my friends that were present in my (social) life during my PhD.

Furthermore, I thank Andrey Rybalchenko for his challenging ideas and suggestions that showed me new directions for further work.

I spent the last month of my PhD in Tsukuba University, in the symbolic computation research group of Tetsuo Ida. I wish to thank Ida-sensei for his offered support and the pleasant working environment in the SCORE laboratory. Also, I would like to thank Mircea Marin for all his help during my stay in Tsukuba.

Finally, I would like to thank my family - Éanya, Éapa, Nagyi, Tata, Levi, Timi, and the little Dóra - for their support and patience through all these years.

Last but not least, I want to express my gratitude to Harald. His patience and understanding through the past years helped me both on the scientific and personal levels. He gave me power and self-confidence in myself and the work I do.

THANK YOU!

Contents

1 Introduction

Writing is nature's way of letting you
know how sloppy your thinking is.
– Guindon –

Program (algorithm) verification has a long research tradition but so far have had relatively little impact on software development in industry (Buchberger, 2003). However, the design and implementation of reliable software remains to be an important issue and any progress in this area will be of utmost importance for the future of the software industry. A challenge for computer science is thus to develop methods that ensure program correctness, hence increasing functionality and reliability of software.

The key method for ensuring program correctness is verification. The idea of verification is to prove program correctness formally. Correctness means that the program satisfies its specification. In this thesis we assume that the specification is given by two logical formulas representing the *precondition* and the *postcondition* of the program. The precondition characterizes the set of initial states in which the program is started, whereas the postcondition describes the set of desirable final or output states, where a *state* is a mapping that maps every variable to a value from a domain corresponding to the type of this variable.

One of the most rigorous way to imperative program verification is based on versions of *Hoare logic* (Hoare, 1969). In this logic for some programs one can compute either the weakest precondition or the strongest postcondition as presented in (Floyd, 1967; Dijkstra, 1976; Winskel, 1994). Such computations allow us to capture the semantics of small program components and in the process derive specifications that can be used to assess whether a given program fragment has the required functionality.

According to (Buchberger, 2003), approaching the problem of *imperative program verification* from a practical point of view has certain implications concerning the following: the style of specifications, the programming language which is used, the help provided to the user for finding appropriate loop invariants, the theoretical framework used for formal verification, the language used for expressing generated verification theorems as well as the database of necessary mathematical knowledge, and finally the proving power, style and language. The *Theorema* system (www. theorema.org) (Buchberger *et al.*, 2006) has certain capabilities which make it appropriate for such a practical approach, given that it is built on top of the computer algebra system *Mathematica* (Wolfram, 2003), has a theorem prover for first-order logic able to perform inductive reasoning, domain specific reasoning, solving and computing.

Programs written in the functional style can be expressed directly in the *Theorema* language, thus the "compilation" step (and its possible errors) is avoided. However, for users who are more comfortable with the imperative style, we also implemented in *Theorema* a simple imperative

programming environment with an interpreter and a verifier. We are able thus to integrate in the system the verification of *procedural (or imperative)* programs (Kirchner, 1999; Kovács and Jebelean, 2003b, 2004a), by using a *verification condition generator* based on *Hoare logic* (Hoare, 1969) and the weakest precondition method (Floyd, 1967; Dijkstra, 1976). This verification tool provides readable arguments for the correctness of programs, with useful hints for debugging.

The logically deep parts of the code are characterized by (possibly nested) loops or recursions. These parts constitute the "tricky" or interesting part of the code. For them, formal program verification *is* an appropriate tool. Verification of loops needs additional information, so-called annotations, expressing conditions at certain intermediate points of the program. They describe relationships between variables which are meant to "hold" during execution. This means that, whenever the program execution passes through such a point, the assertion should evaluate to `True`.

Assertions could be inserted at any point of the program, but in fact they are only necessary in loops as so-called loop invariants. During execution, loop invariants have to evaluate to `True` before and after each iteration. A loop may have many invariants. An appropriate invariant is an assertion that captures all relevant invariant properties of a loop.

Annotating a program is often a non-trivial task and needs a good understanding of how the algorithm works. The idea of the invariants is mostly identical to the basic design idea, and therefore most of the properties established during imperative program verification are either loop invariants or those heavily relying upon the invariants. The effectiveness of automated formal verification thus crucially depends on whether loop invariants, even trivial one, can be deduced automatically. It is agreed (Futschek, 1989) that finding such annotations automatically is in general impractical — thus most systems will just ask the user for the appropriate assertion (see e.g. the Spark system (Barnes, 2003)). However, in many practical situations finding the expression or at least giving some useful hints is quite feasible. This may also be very helpful to the user, since explicitly stated program invariants can help them to identify program properties that must be preserved when the code is to be modified.

In this thesis we present a method that, by combining advanced techniques from algorithmic combinatorics and polynomial algebra, automatically infers *polynomial invariants* without any additional interaction with users.

Polynomial identities found by an automatic analysis are useful for program verification, as they provide non-trivial valid assertions about the program, and thus significantly simplify the verification task. Finding valid polynomial identities (i.e. invariants) has applications in many classical data flow analysis problems (Müller-Olm *et al.*, 2006), such as constant propagation, discovery of symbolic constants, discovery of loop induction variables, etc.

By analyzing the body of the loop, we can try to express the values of the loop variables at the current loop iteration in terms of their previously computed values. In other words, the assignment statements from a loop body form a system of *recurrence equations* describing the behavior of loop variables. A *solution to these recurrences*, that is a closed form solution of the loop variables, would allow us to express their values as functions of the loop iteration. Hence, finding closed form solutions for recurrence equations can be considered a major challenge in reasoning about imperative loops, for automatically inferring invariant relations among the loop variables.

Following this idea, the key steps of our method for invariant generation are as follows.

(i) Assignment statements from the loop body are extracted. They form a system of *recurrence equations* describing the behavior of those loop variables that are changing at each loop iteration.

(ii) Methods of algorithmic combinatorics are used to *solve exactly* the recurrence equations, yielding the *closed form* for each loop variable.

(iii) *Algebraic dependencies* among possible exponential sequences of algebraic numbers occurring in the closed forms of the loop variables are derived using algebraic and combinatorial methods.

As a result of these steps, every program variable can be expressed as a polynomial of the initial values of variables (those when the loop is entered), the loop counter, and some new variables, where there are algebraic dependencies among the new variables.

(iv) Loop counters are then eliminated by polynomial methods to derive a finite set of polynomial identities among the program variables as invariants. From this finite set, under additional assumptions when the loop body contains conditionals branches, any polynomial identity that is a loop invariant can be derived.

In our approach to invariant generation, a family of imperative loops, called *P-solvable* (to stand for polynomial-solvable), is identified, for which test conditions in the loop and conditional branches are ignored and the value of each program variable is expressed as a polynomial of the initial values of variables, loop counter, and some new variables where there are algebraic dependencies among the new variables. We show that for such loops, polynomial invariants can be automatically generated.

Further, if the bodies of these loops consist only of assignments whose right hand sides are polynomials of certain shape, then the approach generates a *complete* set of polynomial invariants of the loop from which any other polynomial invariant can be obtained.

Moreover, if the P-solvable loop bodies contain conditional branches as well, under additional assumptions the approach is proved to be also *complete* in generating a set of polynomial invariants of the loop from which any further polynomial invariant can be derived. We could not find any example of a P-solvable loop with conditional branches and assignments for which our approach fails to be complete. We thus conjecture that the imposed constraints cover a large class of imperative programs, and the completeness proof of our approach without the additional assumptions is a challenging task for further research.

Many non-trivial algorithms working on numbers can be naturally implemented using P-solvable loops.

◇◇◇◇◇

Research into methods for automatically generating loop invariants has a long history, starting with the works of German and Wegbreit (1975) and Karr (1976). However, success was somewhat limited to cases where only few arithmetic operations (mainly additions) among program

variables were involved. Recently, due to the increased computing power of hardware, as well as advances in methods for symbolic manipulation and automated theorem proving (Buchberger, 1996; Robinson and Voronkov, 2001), the problem of automated invariant generation is once again getting considerable attention (Bensalem *et al.*, 1996; Bjørner *et al.*, 1997), based essentially on two main approaches, namely, using either *static* or *dynamic* techniques for invariant discovery.

A *dynamic method* executes a program on a collection of inputs and infers invariants from captured variable traces. In other words, dynamic invariant detection runs a program, observes the values that the program computes, and then reports properties that were true over the observed executions. Thus, the accuracy of the inferred invariant depends in part on the quality and completeness of the test cases; additional test cases might provide new data from which more accurate invariants can be inferred. Such a system is `Daikon` (Ernst *et al.*, 2001), a prototype invariant detector that instruments the source program to trace the variables of interest and runs the instrumented program over a set of test cases in order to infer invariants over the instrumented variables and over derived variables not appearing in the original program. Another system, `Andrew` (Andrews, 1998), generates invariants by comparing the actual behavior of the program against of a user-defined model and indicating divergences between the two. Cook and Wolf (1998) use event traces, which describe the sequence of events in a possibly parallel system, to produce a finite state machine generating the state. They use static and other techniques to detect sequencing, conditionals and iteration.

Contrarily to the dynamic approach, the *static method* of invariant generation operates on the program text, not on the test runs, therefore it has the advantage that the reported properties are true for any program run. Our work belongs to this particular branch of invariant generation. Theoretically, by the static approach one can detect sound invariants, particularly by applying methods from *abstract interpretation* (Cousot and Halbwachs, 1978) and *polynomial algebra* (Buchberger, 2006, 1985).

Using the ideas from iterative techniques for invariant generation, under the framework of *abstract interpretation* (Cousot and Cousot, 1976, 1977a; Cousot and Halbwachs, 1978), in (Tiwari *et al.*, 2001) a method for invariant generation and strengthening is presented. First, an abstract description of the process of inductive invariant generation and strengthening based on computing under- and over-approximations of the reachable state set is given. Then restricted widening, narrowing and quantifier elimination are performed.

Recently, based on the abstract interpretation framework, many researchers (Müller-Olm and Seidl, 2002; Rodriguez-Carbonell and Kapur, 2004; Sankaranaryanan *et al.*, 2004; Kapur, 2006) proposed methods for automatically computing polynomial invariant identities using algorithms from the theory of polynomial ideals.

In (Müller-Olm and Seidl, 2002; Sankaranaryanan *et al.*, 2004) a method built upon polynomial ideal theory and using *Gröbner basis* (Buchberger, 2006, 1985) is presented. This method translates the invariant generation problem into a *constraint solving* one.

In (Sankaranaryanan *et al.*, 2004), non-linear polynomial invariants are proposed as templates with parameters. Constraints on parameters are generated by forward propagation such that the inductiveness of the assertion is guaranteed, and finally, these constrains are solved using the theory of ideals over polynomial rings. Their technique does not rely on widening.

In (Müller-Olm and Seidl, 2002) backward propagation is performed for non-linear programs (programs with non-linear assignments) without branch conditions. Using techniques from abstract interpretation and computer algebra (Hilbert's Basis Theorem and Buchberger's Gröbner basis algorithm), without applying widening/narrowing methods, they generate polynomial invariants by computing a polynomial ideal that represents the weakest precondition for the validity of a *generic polynomial relation* at the target program point. All the conditionals are abstracted to non-deterministic choices.

Both approaches generate polynomial invariants, but they need to fix a priori the degree of a generic polynomial template. This is also the case in (Kapur, 2006), where a method for invariant generation using *quantifier-elimination* (Collins, 1975; Enderton, 1992) is proposed. A *parameterized* invariant formula at any given control point is hypothesized and constraints on parameters are generated by considering all paths through that control point. Solutions of these constraints on parameters are then used to substitute for parameters in a parameterized invariant formula to generate invariants. Non-linear constraint solving is also applied in (Colon *et al.*, 2003) for finding invariant linear *inequalities*.

A related approach for polynomial invariant generation without any a priori bound on the degree of polynomials is presented in (Rodriguez-Carbonell and Kapur, 2004). It is observed that polynomial invariants constitute an ideal. Thus, the problem of finding all polynomial invariants reduces to computing a finite basis of the associated polynomial invariant ideal. This ideal is approximated using a fix-point procedure by computing iteratively the Gröbner basis of a certain polynomial ideal. The fixed point procedure is shown to terminate when the list of (conditional) assignments present in the loop constitutes a *solvable mapping*.

In our work we do not need to fix a priori the degree of a polynomial assertion, and do not use the abstract interpretation framework either. Instead, recurrence relations expressing the value of each program variable at the end of any iteration are formulated and solved exactly. Structural conditions are imposed on recurrence relations so that their closed form solutions can be obtained by advanced symbolic summation techniques. Since these closed form expressions can involve exponentials of algebraic numbers, algebraic dependencies among these exponentials need to be identified, which can be done automatically, contrarily to (Rodriguez-Carbonell and Kapur, 2004), where polynomial dependencies could be derived only for a special case of algebraic exponentials, namely, for rationals. Finally, for eliminating the loop counter and the variables standing for the exponential sequences in the loop counter from these closed form solutions expressed as polynomials, a Gröbner basis computation is performed; however, unlike (Rodriguez-Carbonell and Kapur, 2004), we do not need to perform Gröbner basis computations iteratively. Contrarily to (Rodriguez-Carbonell and Kapur, 2007b) where completeness is always guaranteed, the completeness of our method for loops with conditional branches and assignments is proved only under additional assumptions over ideals of polynomial invariants. It is worth to be mentioned though that these additional constraints cover a wide class of loops, and we could not find any example for which the completeness of our approach is violated.

Our approach is very much in the spirit of the earlier works (Elspas *et al.*, 1972; Wegbreit, 1974; Katz and Manna, 1976) from the 1970s.

◇◇◇◇◇

The thesis is structured as follows.

In Section 2 the problem of program verification is addressed and motivated. The general approach for imperative program verification by using Hoare logic is discussed, followed by a presentation of the weakest precondition strategy for reasoning about imperative programs.

After a brief overview of the *Theorema* system, Section 3 contains the presentation of the imperative verification and programming environment in *Theorema*.

Section 4 introduces the reader to the theoretical notions and properties of the algebraic techniques that are further used in the thesis. Several results of polynomial ideal theory, symbolic summation and algorithmic combinatorics are discussed and presented on examples, with references to relevant related works.

Section 5 starts with introducing the syntax and semantics of the considered programs and presenting ideal properties of polynomial invariants. Then a class of loops with only assignments and ignored test conditions, called P-solvable, is defined for which the closed forms of loop variables are polynomial expressions in the initial values of variables, the loop counter and algebraically related exponential sequences. Further, our algorithm for invariant generation for P-solvable loops with only assignments is presented, followed by proofs of soundness and completeness of the algorithm. By these results, we show that our approach based on algebraic techniques treats in a complete manner a special class of P-solvable loops, called the affine loops. After a detailed discussion on related work, the section ends with several examples on which the implementation of our method is successfully applied. Moreover, for some textbook examples implementing nontrivial algorithms working on numbers, we also describe how their verification in the *Theorema* system can be automatically achieved.

Section 6 addresses the problem of invariant generation for loops with conditional branches and assignments. First, the class of P-solvable loops with conditional branches and ignored test conditions is defined. Then our algorithm for polynomial invariant generation is described and proved to be correct. Further, for some special cases, the approach is proved to be also complete. Finally, the class of loops with conditional branches and affine assignments is briefly discussed. At the end of the section, the steps of the algorithm are presented on several examples. For all these examples, we also present how the automatically inferred invariants can be further used for imperative verification in *Theorema*.

We implemented our method in a new software package `Aligator`, on top of the computer algebra system *Mathematica*. `Aligator` stands for **a**utomated **l**oop **i**nvariant **g**eneration by **a**lgebraic **t**echniques **o**ver the **r**ationals and supports reasoning about loops. We successfully tried it on many programs working on numbers. Section 7 contains a brief description of `Aligator`, with illustrating examples. All examples of imperative loops presented in the thesis, for which polynomial invariants are needed to be verified, are treated automatically by `Aligator`.

The last section of the thesis, Section 8, contains conclusions and some ideas for further work.

2 Reasoning about Imperative Programs

One of the most common approaches to imperative program verification is *axiomatic reasoning*, initiated by R. W. Floyd (Floyd, 1967) for the verifications of flowcharts, and developed further by T. Hoare (Hoare, 1969) into a syntax-directed approach dealing with while-programs. The approach relies on assertions that are attached to the program at several control points. Using these assertions, one can relate the program to its semantics, in other words to its desired behavior expressed by the specification of the program.

With this approach, firstly a language that makes it possible to specify the relevant program properties is needed. The language of first-order predicate logic is appropriate for this type of reasoning. Moreover, the concept of proof system (a set of axioms and proof rules) may be used for deriving that a given program satisfies its specification. Such a proof will proceed in a syntax-directed manner by induction on the structure of the program.

T. Hoare's method received high attention, and many Hoare-style proof systems have been proposed since then in order to make possible computer-aided automated program verification, using *verification condition generators*. A verification condition generator implements inference rules for translating a formally specified and annotated program into predicate logic formulas, which are to be proved in order to prove correctness of programs.

2.1 Program, Specification and Correctness

A *program specification* must describe exactly what execution of a program is to accomplish, while a *program* describes how a certain input state is transformed into a desired output state. (Another part of the specification might also deal with size, speed and another properties.)

An *imperative program* is a finite sequence of *statements*, where a *statement* will denote single commands of programming languages, such as: empty statements, assignments, conditionals, loops.

To *specify* a program with its specification, we rely on the so-called *Hoare triple* from the Hoare axiom system (Hoare, 1969), introduced by T. Hoare. A Hoare triple has a form

$$\{P\}\ S\ \{Q\}, \tag{2.1}$$

where

- S is a program;
- P is a logical formula called the *precondition* or *input assertion* of S;
- Q is a logical formula called *postcondition* or *output assertion* of S.

The braces $\{$ and $\}$ around the assertions are used to separate the assertions from the program itself.

Any formula of the form (2.1) is called a *correctness formula*.

Example 2.1 Consider the following correctness formula

$$\{x = 0\}\ x := x + 1\ \{x = 1\}.$$

In this formula

- x=0 is the precondition;
- x := x+1 is the program;
- x=1 is the postcondition.

Example 2.2 Given two natural numbers x and y, with y being a nonzero natural number, compute the quotient *quo* and the remainder *rem* of the division of x by y.
The *precondition* of the program that computes the *rem* and *quo* is expressed by the fragment "given two natural numbers x and y, with y being a nonzero natural number". Assuming the domain of values is $\mathbb{N}$, the precondition can be written using predicate logic as

$$(x \geq 0) \wedge (y > 0).$$

The postcondition of the program is expressed by the fragment "the quotient *quo* and the remainder *rem* of the division of x by y". Using predicate logic and quotient and remainder properties, the postcondition can be written as

$$(quo * y + rem = x) \wedge (0 \leq rem) \wedge (rem < y).$$

A possible imperative program code for computing the quotient and remainder of the division of x by y is given below.

$$\begin{array}{l} quo := 0;\ rem := x; \\ \texttt{While}[y \leq rem, \\ \qquad rem := rem - y; \\ \qquad quo := quo + 1]. \end{array}$$

Hence, the correctness formula for Example 2.2 is as follows.

$$\{(x \geq 0) \wedge (y > 0)\} \quad \begin{array}{l} quo := 0;\ rem := x; \\ \texttt{While}[y \leq rem, \\ \qquad rem := rem - y; \\ \qquad quo := quo + 1] \end{array} \quad \{(quo * y + rem = x) \wedge (0 \leq rem) \wedge (rem < y)\}.$$

Informally, a program is correct if it satisfies the intended input/output relation, i.e. its specification. There are two interpretations of correctness, see e.g. (Apt and Olderog, 1997; Kirchner, 1999).

- A correctness formula $\{P\}\ S\ \{Q\}$ is true in the sense of *partial correctness* if every *terminating computation* of S that starts in a state satisfying P terminates in a state satisfying Q.
- A correctness formula $\{P\}\ S\ \{Q\}$ is true in the sense of *total correctness* if every computation of S that starts in a state satisfying P *terminates* and its final state satisfies Q.

Thus, in the case of partial correctness, diverging computations of S are not taken into account.

Throughout this thesis, if not mentioned otherwise, we consider partial correctness of imperative programs.

It is well known that the challenging part of the Floyd-Hoare method is the reasoning about programs with loop statements, when additional assertions are needed to express the behavior of the loop and to thus to generate the verification conditions. In the case of partial correctness, these assertions are called *loop invariants*, fulfilling the "inductiveness property", i.e. during program execution loop invariants have to evaluate to `True` *before* and *after* each iteration. Informally, an invariant expresses constant or unchanged loop properties.

2.2 Loop Assertions: Invariants and Termination Terms

Loop invariants are the key to deductive verification of programs, playing a central role in program development. They can protect a programmer from making changes that violate assumptions upon which program correctness depends. The absence of explicitly stated invariants makes it easy for programmers to introduce errors while making changes. Therefore, automatically checking and finding invariants are crucial in the analysis and verification of sequential programs.

Informally, a loop invariant is an assertion that is valid before and after each loop iteration. In particular, it is valid before the loop is entered and after the loop is exited. Formally, it can be defined using the Hoare triple notation as follows.

Definition 2.3 Let b be a boolean expression, treated as a formula.

An assertion I is an *invariant* for the Hoare triple $\{P\}$`While`$[b,\ S]\{Q\}$ if it satisfies the following conditions:

(1) initial condition: $P \Rightarrow I$;

(2) iterative condition: $\{I \wedge b\}\ S\ \{I\}$;

(3) final condition: $I \wedge \neg b \Rightarrow Q$.

Example 2.4 An appropriate loop invariant for Example 2.2 would be

$$(quo * y + rem = x) \wedge (0 \leq rem) \wedge (0 < y) \wedge (x \geq 0).$$

In general, a loop may has many invariants. Among all invariants we are normally interested in those that help us to prove the desired program properties.

To verify total correctness, it is necessary to prove that every program statement terminates, in particular the iterative ones like while loops. For doing so, it is sufficient to find an expression, called *termination term* (or *ranking function*), with the following properties:

- it is expressed in terms of those program variables that change with every iteration of the loop,
- its value is always a natural number that decreases with every single iteration,

and then prove an assertion showing that this function indeed decreases with every loop iteration.

Example 2.5 For the program of Example 2.2, one can use *rem* as a termination term.

Note however that integer-valued termination term (or ranking function) sometimes is not enough and termination is often proved by more sophisticated methods (Cook *et al.*, 2005, 2006).

2.3 Reasoning Principle: Weakest Precondition Strategy

For reasoning about imperative programs we use a simple formal calculation involving a version of Hoare logic, namely, the *weakest precondition strategy* (Dijkstra, 1976).

Definition 2.6 A formula P is said to be *weaker* than a formula R if $R \implies P$ holds.

Hence, if formula P is weaker than R, then P describes more possible states than R.

Weakest precondition calculations (based on Hoare logic) are useful when we have formal postcondition specifications available for the code we want to verify.

Assume that we want to verify a program where we know the postcondition but not the precondition:

$$\{?\}\ S\ \{Q\}.$$

In general there could be arbitrarily many preconditions P which are valid for the program S and postcondition Q. However, there is a *single* precondition which describes the set of *all* possible initial states such that the execution of S would lead to a state satisfying Q. This precondition is called the *weakest precondition*.

Definition 2.7 Let Q be an assertion and S be a program.

An assertion P is called the *weakest precondition* for S and postcondition Q if for every assertion R such that

$$\{R\}\ S\ \{Q\}$$

we have

$$R \implies P.$$

We denote the weakest precondition for a program S and a postcondition Q by $\mathrm{wp}(S,Q)$.

In other words, $wp(S,Q)$ is a predicate describing the set of all initial states that will guarantee termination of S in a state satisfying Q, and we thus have

$$\{\mathrm{wp}(S,Q)\}\ S\ \{Q\}.$$

For verifying $\{P\}\ S\ \{Q\}$ using the weakest precondition strategy, we first determine

$$\mathrm{wp}(S,Q),$$

and then prove

$$P \implies \mathrm{wp}(S, Q).$$

Unfortunately, when a program S contains loops, the weakest precondition may be not expressible in the assertion language. Even when it is expressible, it may not be easy to find. For this reason, for verification of programs with loops, in addition to weakest preconditions one uses loop invariants.

The function wp for one certain statement can be viewed as a function which only takes a predicate (the postcondition) and returns another predicate (the weakest precondition). Therefore wp is also called a *predicate transformer*.

However, in the process of generating the weakest precondition $\mathrm{wp}(S, Q)$ also other formulas are generated, that are called *verification conditions* (vc). Thus, for verifying $\{P\}\ S\ \{Q\}$, one needs to prove also the correctness of these verification conditions in order to ensure correctness of program. For this reason, throughout the thesis, verification conditions are also called *proof obligations*.

For a composition $S = s_1; ...; s_n$ of statements, the function wp works recursively backwards, statement-by-statement, on the program syntax, in such a way that at the end the program code is "eliminated" and the weakest precondition is obtained together with additionally generated verification conditions, as pictured below.

Weakest Precondition Strategy:

$$
\begin{array}{ll}
\{P\} & \\
 & \leftarrow \mathrm{wp}(s_1,\ \mathrm{wp}(s_2, \mathrm{wp}(..., \mathrm{wp}(s_{n-1}, \mathrm{wp}(s_n, Q))...))) \\
s_1; & \\
 & \leftarrow \mathrm{wp}(s_2,\ \mathrm{wp}(s_3, \mathrm{wp}(..., \mathrm{wp}(s_{n-1}, \mathrm{wp}(s_n, Q))...))) \\
s_2; & \\
\vdots \quad & \vdots \qquad\qquad\qquad\qquad\qquad\qquad \uparrow \bigwedge \text{ verification conditions (vc)} \\
 & \leftarrow \mathrm{wp}(s_{n-1},\ \mathrm{wp}(s_n,\ Q)) \\
s_{n-1}; & \\
 & \leftarrow \mathrm{wp}(s_n,\ Q) \\
s_n & \\
\{Q\} &
\end{array}
$$

Thus, in order to show $\{P\}\ S\ \{Q\}$ by applying the weakest precondition strategy, one has to show the validity of the generated verification conditions (vc), as well as the additional verification condition given by the implication

$$P \implies \mathrm{wp}(s_1,\ \mathrm{wp}(s_2, \mathrm{wp}(..., \mathrm{wp}(s_{n-1}, \mathrm{wp}(s_n, Q))...))),$$

since $\mathrm{wp}(s_1,\ \mathrm{wp}(s_2, \mathrm{wp}(..., \mathrm{wp}(s_{n-1}, \mathrm{wp}(s_n, Q))...))) = \mathrm{wp}(S, Q)$.

The generation of the weakest precondition is done by applying specific inference rules (*predicate transformation rules*) for wp on each language command (Dijkstra, 1976). These rules translate program statements into mathematical formulas.

For further purposes, in this thesis we state only the partial and total correctness verification rule of imperative while loops.

Verification Rule for Partial Correctness of While Loops.

The problem with verifying programs with while loops is that one cannot in general compute the weakest precondition of the loops. To verify such programs one uses loop invariants in place of weakest preconditions. To this end, one should first annotate while loops with assertions intended to serve as invariants and then use these assertions.

Let us denote a while loop $\texttt{While}[b,\ S]$ annotated by an assertion I by $\texttt{While}[b,\ S,\ I]$.

For a while loop with an annotation I, we will use a notion of the weakest precondition different from ordinary while loops. Namely, we define the weakest precondition of such a loop to be I and we thus have

$$\mathrm{wp}(\texttt{While}[b,\ S,\ I],\ Q)\ =\ I,$$

whereas the produced verification conditions for ensuring partial correctness of this loop are

$$\begin{aligned} &I \wedge b \implies \mathrm{wp}(S,I);\\ &(I \wedge \neg b) \implies Q. \end{aligned} \tag{2.2}$$

Proving partial correctness of the Hoare triple $\{P\}\ \texttt{While}[b,\ S,\ I]\ \{Q\}$ thus reduces to proving the verification conditions (2.2) together with the additional verification condition given by the implication $P \implies I$.

Example 2.8 By applying the weakest precondition strategy for partial correctness verification of Example 2.2, using the loop invariant specified in Example 2.4, we obtain the following proof obligations.

$$\begin{aligned} &(rem = x) \wedge (quo = 0) \wedge (x \geq 0) \wedge (y > 0) \implies\\ &(x = rem + y * quo) \wedge (x \geq 0) \wedge (rem \geq 0) \wedge (y > 0); \end{aligned}$$

$$\begin{aligned} &(x = rem + y * quo) \wedge (x \geq 0) \wedge (rem \geq 0) \wedge (y > 0) \wedge (y \leq rem) \implies\\ &(x = (rem - y) + y * (quo + 1)) \wedge (x \geq 0) \wedge (rem - y \geq 0) \wedge (y > 0); \end{aligned}$$

$$\begin{aligned} &(x = rem + y * quo) \wedge (x \geq 0) \wedge (rem \geq 0) \wedge (y > 0) \wedge \neg(y \leq rem) \implies\\ &(x = rem + y * quo) \wedge (0 \leq rem < y). \end{aligned}$$

Verification Rule for Total Correctness of While Loops.

To verify total correctness of a while loop, the loop is annotated not only by an invariant, but also by a termination term and an additional verification condition is then added whose validity ensures termination. Let us denote such an annotated loop by $\texttt{While}[b,\ S,\ I,\ T]$, where I is an invariant and T a termination term. We assume that T is syntactically guaranteed to be integer-valued.

The weakest precondition of such an annotated while loop is its invariant. Namely, we have:

$$\mathrm{wp}(\texttt{While}[b,\, S,\, I,\, T],\, Q) \;=\; I,$$

whereas the produced verification conditions for ensuring total correctness of the while loop are

$$\begin{aligned} &I \wedge b \implies T \geq 0; \\ &I \wedge b \wedge (T = n) \implies \mathrm{wp}(S, I \wedge (T < n)); \\ &(I \wedge \neg b) \implies Q, \end{aligned} \tag{2.3}$$

where n is a new constant, intended to denote the value of T before the loop (iteration).

Proving total correctness of the Hoare triple $\{P\}\ \texttt{While}[b,\, S,\, I,\, T]\ \{Q\}$ thus reduces to proving the verification conditions (2.3) together with the additional verification condition given by the implication $P \implies I$.

Example 2.9 By applying the weakest precondition strategy for total correctness verification of Example 2.2, using the loop invariant specified in Example 2.4 and the termination term specified in Example 2.5, we obtain the following proof obligations.

$$\begin{aligned} &(rem = x) \wedge (quo = 0) \wedge (x \geq 0) \wedge (y > 0) \implies \\ &(x = rem + y * quo) \wedge (x \geq 0) \wedge (rem \geq 0) \wedge (y > 0); \end{aligned}$$

$$\begin{aligned} &(x = rem + y * quo) \wedge (x \geq 0) \wedge (rem \geq 0) \wedge (y > 0) \wedge (y \leq rem) \wedge (rem = n) \implies \\ &(x = (rem - y) + y * (quo + 1)) \wedge (x \geq 0) \wedge (rem - y \geq 0) \wedge (y > 0) \wedge (rem - y < n); \end{aligned}$$

$$(x = rem + y * quo) \wedge (x \geq 0) \wedge (rem \geq 0) \wedge (y > 0) \wedge (y \leq rem) \implies (rem \geq 0);$$

$$\begin{aligned} &(x = rem + y * quo) \wedge (x \geq 0) \wedge (rem \geq 0) \wedge (y > 0) \wedge \neg(y \leq rem) \implies \\ &(x = rem + y * quo) \wedge (0 \leq rem < y). \end{aligned}$$

As shown also in the above examples, in the *verification process* one proves the consistency between specification and program for every possible input, ensuring that the program meets its specification.

With program verification one can gain deep insight into the program, understand the idea behind it and its limitations (Buchberger and Lichtenberger, 1981), and thus program verification supports goal-oriented program development (Gries, 1981). The annotations for program verification are a good documentation for the program (Gries, 1981). However, annotating programs can be very difficult. The difficulty comes from the program size and structure, the language in which the assertions are represented, etc., and therefore verification is usually performed only on a small-scale algorithmic level.

3 Imperative Program Verification in *Theorema*

3.1 The *Theorema* system

The *Theorema* system (Buchberger *et al.*, 2006) (`www.theorema.org`) is a computer mathematical assistant which is implemented on top of the computer algebra system *Mathematica* (Wolfram, 2003). The system allows the definition and the organization of mathematical theories (including the description of algorithms) in the language of higher-order predicate logic, and also offers the necessary environment for experimenting with algorithms (computing) and for investigating the properties of mathematical structures (proving, solving) labeled by intuitive keywords, e.g. "Definition", "Theorem", "Proposition", "Algorithm", etc. Moreover, *Theorema* allows the introduction of ones own notations and symbols, as well as creating new graphical symbols (Nakagawa, 2002).

Theorema aims at supporting the entire process of mathematical theory exploration: invention of mathematical concepts, invention and verification (proof) of propositions about concepts, invention of problems formulated in terms of concepts, invention and verification (proof of correctness) of algorithms solving problems, and storage and retrieval of the formulas invented and verified during this process. This integral objective of supporting the entire mathematical invention and verification process was already formulated at the very beginning of the *Theorema* project (Buchberger, 1996).

Currently the computing and solving capabilities are imported from the underlying computer algebra system *Mathematica*, and are quite powerful, thus the most significant and interesting part of the *Theorema* system is the one providing proving capabilities. Proving is implemented in the system by ways of various domain–specific provers (propositional, first–order predicate logic, limit domains, induction of natural numbers and over lists, proving over sets, proving equalities by the Knuth–Bendix completion, etc.). The general approach is to implement inference rules, as well as strategies and techniques which are similar to the human style of proving. The produced proofs are explained in natural language and are human–readable, meaning that even failed proofs are useful since they may give hints for finding errors or omissions in the respective theory. Although the main emphasis of *Theorema* is on fully automatic proving, the system also has interactive facilities for allowing the user to manually influence the behavior of the provers (Piroi, 2004).

Programs written in the functional style can be expressed directly in the *Theorema* language with equalities interpreted as rewrite rules, thus the "compilation" step (and its possible errors) is avoided. However, for users who are more comfortable with the procedural style, we enhanced the system with an imperative language with interpreter and verifier allowing imperative program verification by generating and proving verification conditions depending on the program syntax

(Kirchner, 1999; Kovács, 2004b).

The *Theorema* system is particularly appropriate for program verification, because it has access to a wealth of powerful computing and solving algorithms from *Mathematica* and it delivers proofs in a natural language by using natural style inferences.

3.2 Imperative Programming Environment in *Theorema*

In *Theorema* a simple but representative imperative language is supported (Kirchner, 1999), in which a program has a specification and an implementation part (the body of the program).

Specifications, invariants and conjectures can be expressed in the logical language of *Theorema*. The implementation and the verification process for imperative programs is done in a prototype verification condition generator integrated in the overall framework of the *Theorema* system.

Abstract Syntax

The basic model of the programming language is quite general. We want to be able to represent a sequential imperative programming language, in which programs are considered as procedures, without return values and with input, output and/or transient parameters.

The used syntactic sets are listed below.

- Ring of numbers $\mathfrak{R}$.
- Truth values $T = \{\texttt{True}, \texttt{False}\}$.
- Identifiers *IdSet*.
- Variables *VarSet*.
- Arithmetic expressions *Aexp*.
- Boolean expression *Bexp*.
- Set of commands *Com*.

Expressions are standard *Theorema* (thus *Mathematica*) boolean and arithmetic expressions. In the process of reasoning about programs, the boolean expressions are treated as formulas and they are also called *assertions*.

The commands of the programming language (Kirchner, 1999; Jebelean *et al.*, 2004) are as follows.

- $x := a$: Assignment, where x is a variable and a is an arithmetic expression (that might also contain a function call).
- $s_1; s_2$: Sequential composition of commands s_1 and s_2.
- $\texttt{IF}[b, s_1, s_2]$: Conditional statement. Depending on the value of the boolean expression b, either s_1 (in case of b is true) or s_2 (in case of b is false) is executed.

- `WHILE`[b,S, optional: `Invariant`, `Assert`,`TerminationTerm`]: Loop command, where S is a sequence of commands representing the loop body, whereas b is a boolean expression representing the loop condition. Optionally, we introduced three constructors that are relevant in the verification process of the loop, namely:

  - `Invariant`, for specifying the complete invariant property of the loop;

  - `Assert`, for specifying only the non-polynomial invariant property of the loop;

  - `TerminationTerm`, for specifying the termination term of the loop.

- `FOR`[*counter*, *lowerBound*, *upperBound*, *step*,S, optional: `Invariant`, `Assert`]: Loop command, where S is the loop body, whereas *counter*, *lowerBound*, *upperBound* are integer variables denoting the loop counter with its minimal and maximal values. The two optional constructors `Invariant` and `Assert` have the same significance as in the case of the `WHILE` loop.

- P[*inVars*,*transVars*,*OutVars*]: Procedure call, where *inVars*, *transVars*, *OutVars* are respectively the lists of input, transient and output variables.

Recursivity and mutually recursive procedure calls are not yet handled.

As showed above, the identifiers of the commands from this imperative programming language integrated in *Theorema* are written with capital letters, suggesting that these statements are not the statements from *Mathematica*, although the semantics and execution of each statement is identical to the well-known *Mathematica* statements. The major difference consists in the fact that, in this imperative language, for the `WHILE` and `FOR` statements we allow optional arguments encoding loop assertions which are relevant in the verification process of the loop.

Basic Interface Constructs of the Programming Language

The user interface of the imperative programming environment in *Theorema* has four main constructs with the following structure.

- *Specification* of the program.

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

  that contains a pre– and a postcondition (specified by `Pre` and `Post`) and also an interface definition. An interface definition consists of the program name (interface name) with its parameters by indicating also their nature: input $\downarrow$, output $\uparrow$, input-output/transient $\updownarrow$. (We do not yet support data types.)

  Using this imperative programming environment of *Theorema*, the specification of Example 2.2 can be written as follows.

  Example 3.1

$$\begin{aligned}&\texttt{Specification}[\text{"}\textit{Division}\text{"},\ \textit{Div}[\downarrow x,\downarrow y,\uparrow \textit{rem},\uparrow \textit{quo}],\\ &\qquad \texttt{Pre} \rightarrow ((x \geq 0) \wedge (y > 0)),\\ &\qquad \texttt{Post} \rightarrow ((\textit{quo} * y + \textit{rem} = x) \wedge (0 \leq \textit{rem} < y))].\end{aligned}$$

- *Body* of the program.

  `Program[label,interface,code],`

  that contains the program code and also an interface definition. Programs are annotated with invariants and termination terms, and are considered as procedures without return values. We consider non-recursive deterministic programs, and in the program text we differentiate between assignment ":=" and logical equality "=".

  For Example 2.2 , the annotated program code written in this procedural programming environment is presented below.

  Example 3.2

  `Program`["*Division*", $Div[\downarrow x, \downarrow y, \uparrow rem, \uparrow quo]$,
  $quo := 0; rem := x;$
  `WHILE`$[y \leq rem,$
  $rem := rem - y;$
  $quo := quo + 1,$
  `Invariant` $\rightarrow ((quo * y + rem = x) \wedge (0 \leq rem) \wedge (0 < y) \wedge (x \geq 0)),$
  `TerminationTerm` $\rightarrow rem]]$.

- *Interpreter*, for testing the program body with concrete input values.

  `Execute[interface_name[concrete_values]].`

  For Example 2.2, a particular testing with its result is as follows.

  Example 3.3

  `Execute`$[Div[65,\ 17,\ rm,\ qt]];\ \{rm,\ qt\},$

  with the output

  $$\{14,\ 3\}.$$

- *Verifier* (verification condition generator).

  `VCG[Program[label],Specification[label]].`

  The `VCG` takes a specification and program code and produces the verification conditions of the program based on the weakest precondition strategy.

Verification of imperative programs in *Theorema* contains two main parts: generation and proving verification conditions.

Generation of Verification Conditions

We rely on the axiomatic semantic for imperative programs written in terms of the Hoare triple (Hoare, 1969)

$$\{P\}S\{Q\},$$

where $P, Q \in BExp$ and are treated as formulas, and $S \in Com$.

More specifically, based on the *weakest precondition strategy* (see Subsection 2.3), a package for imperative program verification, called *Verification Condition Generator (*`VCG`*)* (Kirchner, 1999; Kovács, 2004b), is integrated in *Theorema*. `VCG` takes an annotated imperative program with its specification, and produces as output a *Theorema* lemma as a collection of the universally quantified verification conditions.

`VCG` is a predicate translator based on a list of inference rules. It is recursive on the structure of the code and works back-to-front, statement-by-statement. Internally it repeatedly modifies the postcondition using a predicate transformer such that at the end the result is the verification condition

$$\texttt{P} \Longrightarrow \texttt{transformed postcondition},$$

together with a list of verification conditions. `VCG` is thus a function:

$$\texttt{VCG}: \langle \texttt{S, (P,Q)} \rangle \rightarrow \textit{Theorema}\ \text{lemma},$$

and uses another function, called *Extended Predicate Transformer (*`EPT`*)*. `EPT` takes the program and its postcondition, and produces the *weakest precondition (*wp*)* of the program together with additional verification lemmas, as written below.

$$\texttt{EPT}: \langle \texttt{S, Q} \rangle \rightarrow \langle \texttt{wp(S,Q), verification conditions} \rangle.$$

`VCG` uses this list of verification lemmas and adds the implication

$$P \Rightarrow wp(S,Q)$$

to the verification lemmas, providing thus the formulas that are to be proved in order to prove the correctness of the given program w.r.t. its specification.

For Example 2.2, using Examples 3.1 and 3.2, the call of `VCG` and the produced verification conditions in the *Theorema* syntax are as follows.

Example 3.4

$$\texttt{VCG}[\texttt{Program}[\text{“}\textit{Division}\text{”}], \texttt{Specification}[\text{“}\textit{Division}\text{”}]].$$

Lemma (Division) :
for any : x, y, rem, quo
(WHILE.Inv + Term)
$(x = rem + quo * y) \wedge 0 \leq rem \wedge 0 < y \wedge x \geq 0 \wedge y \leq rem \wedge (\mathrm{T1} = rem) \Rightarrow$
$(x = rem + (-1) * y + (1 + quo) * y) \wedge 0 \leq rem + (-1) * y \wedge 0 < y \wedge x \geq 0 \wedge rem + (-1) * y < \mathrm{T1}$
(WHILE.Final)
$(x = rem + quo * y) \wedge 0 \leq rem \wedge 0 < y \wedge x \geq 0 \wedge y \not\leq rem \Rightarrow (rem + quo * y = x) \wedge 0 \leq rem \wedge rem < y$
(WHILE.Term)
$(x = rem + quo * y) \wedge 0 \leq rem \wedge 0 < y \wedge x \geq 0 \wedge y \leq rem \Rightarrow rem \geq 0$
(Init)
$0 \leq x \wedge 0 < y \Rightarrow (x = x) \wedge 0 \leq x \wedge 0 < y \wedge x \geq 0$

`Division` is the label of the lemma, and `WHILE.Term`, etc. are the labels of the individual formulas.

Proving Verification Conditions

The automatically obtained verification conditions are fed into the available provers of the *Theorema* system.

In most cases the `PCS` prover of *Theorema* is applied (Buchberger *et al.*, 2006). `PCS` uses quantifier elimination, and produces human–readable proofs of the verification conditions (i.e. program correctness is proved) or failures in case the program is not correct.

The proving part of the verification process is not in the scope of the thesis. However, it is worth to be mentioned that in many examples the generated verification conditions were proven automatically by calling `PCS`.

Using the labels of formulas from Example 3.4, one may call the `PCS` prover of *Theorema* in order to check the validity of the verification conditions, as presented below.

`Prove[Lemma["`*Division*`"], by → PCS]`.

Further textbook examples illustrating the imperative verification environment of *Theorema* will be presented later in this thesis, in Sections 5 and 6. For these examples implementing interesting algorithms working on numbers, we first generate all their polynomial invariants using our software package `Aligator` implemented in *Mathematica*. The automatically inferred polynomial invariants, together with other (user-asserted) non-polynomial equality invariants, are used for annotating the program code of the example. The annotation process, i.e. integration of `Aligator` in *Theorema* is not yet computer-supported and is expected to be done manually by the user.

The annotated program code is further used for checking partial or total correctness w.r.t. the given specification, by calling the verification condition generator. For each example we show the output of `VCG`, namely, a *Theorema* lemma containing universally quantified first-order formulas representing the verification conditions of the example.

Proving these conditions is beyond the scope of this thesis. However, Subsection 7.9 contains a table with the summary of the results of applying the `PCS` prover of *Theorema* for proving the verification conditions for partial correctness verification of the examples treated in the thesis.

4 Algebraic Considerations

As presented in Sections 2 and 3, the creative and challenging part in imperative program verification based on Hoare logic is to reason about loops. For these cases, additional assertions about loop properties are needed, such as loop invariants and termination terms.

In this thesis algorithms for systematically inferring polynomial relations as loop invariants are presented. For doing so, advanced techniques from polynomial algebra, algorithmic combinatorics and computational logic are applied, as it will be described in Sections 5 and 6.

To make presentation self-contained, in this section we briefly present the algebraic methods, fundamental definitions and properties of linear recurrences, ideals and algebraic dependencies, and also fix some relevant notation. These definitions, properties and notations will be used further in the thesis. The results presented in this section are not original. For additional details, the interested reader can consult (Cox *et al.*, 1998; Everest *et al.*, 2003; Graham *et al.*, 1989).

In what follows, $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}$ denote respectively the set of natural, integer, rational and real numbers. Throughout this thesis we assume that $\mathbb{K}$ is a field of characteristic zero (e.g. $\mathbb{Q}$, $\mathbb{R}$, etc.), and by $\bar{\mathbb{K}}$ we denote its algebraic closure. All rings are commutative.

4.1 Ideals and Gröbner Bases

Since algorithmic computation of polynomial properties will play a major role later in the thesis, in this section we shortly study polynomials in $m \in \mathbb{N}$ variables $x_1, \ldots, x_m$ with coefficients in the arbitrary field $\mathbb{K}$, by defining first the notion of monomials.

Definition 4.1 A *monomial* in $x_1, \ldots, x_m$ is a product

$$x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_m^{\alpha_m},$$

where $\alpha_1, \ldots, \alpha_m \in \mathbb{N}$. The *total degree* of this monomial is the sum $\alpha_1 + \cdots + \alpha_m$.

For simplicity, we denote $X = \{x_1, \ldots, x_m\}$ and $\alpha = \{\alpha_1, \ldots, \alpha_m\} \in \mathbb{N}^m$, and write

$$X^\alpha = x_1^{\alpha_1} \ldots x_m^{\alpha_m}.$$

When $\alpha = \{0, \ldots, 0\}$, then $X^\alpha = 1$.
Also, for $\alpha = \{\alpha_1, \ldots, \alpha_m\} \in \mathbb{N}^m$, we use the notation $|\alpha| = \alpha_1 + \cdots + \alpha_m$.

Definition 4.2 A *polynomial* p in $X = \{x_1, \ldots, x_m\}$ over $\mathbb{K}$ is a (finite) linear combination of monomials with coefficients in $\mathbb{K}$, namely:

$$p = a_1 X^{\alpha_1} + \cdots + a_t X^{\alpha_t},$$

where $t \in \mathbb{N}$, $\alpha_1, \ldots, \alpha_t \in \mathbb{N}^m$ and $a_1, \ldots, a_t \in \mathbb{K}$.
The *ring of polynomials* in X with coefficients in $\mathbb{K}$ is denoted by $\mathbb{K}[X]$, and $\mathbb{K}$ is the coefficient field of $\mathbb{K}[X]$.
The elements of $\mathbb{K}[X]$ are called *rational functions* over $\mathbb{K}$.

Finally, for dealing with polynomials, we will also use the following notions.

Definition 4.3 Let $p = a_1 X^{\alpha_1} + \cdots + a_t X^{\alpha_t}$ be a polynomial in $\mathbb{K}[X]$.

(1) a_i is the *coefficient* of the monomial x^i.
(2) If $a_i \neq 0$, then $a_i x^i$ is a *term* of p.
(3) The *total degree* , denoted by *deg(f)*, is the maximum $|\alpha_i|$ s.t. the coefficient of a_i is nonzero.

Definition 4.4

- A subset $I \subset \mathbb{K}[X]$ is an *ideal* if
  (1) $0 \in I$;
  (2) for all $p_1, p_2 \in I$, we have $p_1 + p_2 \in I$;
  (3) for all $p \in I$ and $q \in \mathbb{K}[X]$, we have $qp \in I$.
  We write $I \trianglelefteq \mathbb{K}[X]$ to denote that I is an ideal in $\mathbb{K}[X]$.
- Let $p_1, \ldots, p_s \in \mathbb{K}[X]$, $s \in \mathbb{N}$. The *smallest ideal* containing $p_1, \ldots, p_s$, denoted by $\langle p_1, \ldots, p_s \rangle$, is
  $$\langle p_1, \ldots, p_s \rangle = \Big\{ \sum_{i=1}^{s} q_i p_i \mid q_1, \ldots, q_s \in \mathbb{K}[X] \Big\},$$
  and we say that $\langle p_1, \ldots, p_s \rangle$ is the ideal *generated* by $p_1, \ldots, p_s$, whereas the set $\{p_1, \ldots, p_s\}$ is a *basis* of $\langle p_1, \ldots, p_s \rangle$.
- Let $I, J \trianglelefteq \mathbb{K}[X]$. Then
  (1) $I + J = \{a + b \mid a \in I, b \in J\}$ is called the *sum* of I and J;
  (2) $I \cdot J = \{a \cdot b \mid a \in I, b \in J\}$ is called the *product* of I and J.
- Let $f,\ g \in \mathbb{K}[X]$ and $I \trianglelefteq \mathbb{K}[X]$. We say that f is *congruent* to g modulo I, and we denote it by $f \equiv g$, if $f - g \in I$.
  "$\equiv$" is an equivalence relation on $\mathbb{K}[X]$. The set of equivalence classes is denoted by $\mathbb{K}[X]/I$, and it is called the *quotient ring* of $K[X]$ by I. The elements of the quotient ring are called *cosets* and they are of the form $f + I$.

By Hilbert's basis theorem (Adams and Loustaunau, 1994), every ideal in $\mathbb{K}[X]$ has a finite basis. This result underlines the practical importance of ideals. Namely, it allows algorithmic computation of polynomial properties by using a special ideal basis called *Gröbner basis*, whose effective computation is possible by the *Buchberger Algorithm* (Buchberger, 2006, 1985).

The theory of Gröbner bases allows to answer algorithmically many questions about ideals, such as ideal membership of a polynomial, equality and inclusion of ideals, etc.

Definition 4.5

(1) A strict linear (or total) ordering $\prec$ on X is *admissible* (or *monomial ordering*) on $\mathbb{K}[X]$ if

- $1 \prec r$ for all monomial terms r in X;
- for any monomial terms r,u,v in X: $u \prec v \Rightarrow ur \prec vr$.

(2) Let $p = a_1X^{\alpha_1} + \cdots + a_tX^{\alpha_t}$ be a nonzero polynomial in $\mathbb{K}[X]$ with $a_t \neq 0$. Let $\prec$ be a monomial order with $X^{\alpha_1} \prec X^{\alpha_2} \prec \cdots \prec X^{\alpha_t}$. Then

- the *leading term* of p is
$$\mathrm{LT}(p) = X^{\alpha_t};$$
- the *leading coefficient* of p is
$$\mathrm{LC}(p) = a_t;$$
- the *leading monomial* of p is
$$\mathrm{LM}(p) = a_tX^{\alpha_t}.$$

(3) A finite subset $G = \{g_1,\ldots,g_t\}$ of an ideal $I \trianglelefteq \mathbb{K}[X]$ is a *Gröbner basis* w.r.t. the ordering $\prec$ if
$$\langle \mathrm{LT}(g_1),\ldots,\mathrm{LT}(g_t)\rangle = \langle \mathrm{LT}(I)\rangle.$$
$G = \{g_1,\ldots,g_t\} \subset I$ is thus a Gröbner basis of I if the leading term of any element of I is divisible by one of the $\mathrm{LT}(g_i)$.

(4) A polynomial $p \in \mathbb{K}[X]$ is *completely reduced* (or *reduced* or *irreducible*) with respect to a set $G \subset \mathbb{K}[X]$ if no monomial term of p is a multiple of $\mathrm{LT}(g),\ g \in G$. Otherwise, it is *reducible*.

As mentioned already, a Gröbner basis is used for studying specific polynomial ideal properties that arise in several applications. However, a Gröbner basis of an ideal I is not unique. A unique representation of (all) Gröbner bases of an ideal is thus needed.

Theorem 4.6 Consider a monomial ordering $\prec$ on the terms of X and $I \trianglelefteq \mathbb{K}[X]$.

(1) There is exactly one Gröbner basis G of I, called *the reduced Gröbner basis*, such that

- $\mathrm{LC}(g) = 1$ for all $g \in G$;
- for all $g \in G$, g is completely reduced w.r.t. $G \setminus \{g\}$, in other words, no monomial of g lies in $\langle \mathrm{LT}(G \setminus \{g\})\rangle$.

(2) Let $G = \{g_1,\ldots,g_t\}$ be a Gröbner basis for $I \trianglelefteq \mathbb{K}[X]$, and let $p \in \mathbb{K}[X]$. Then there is a unique $r \in \mathbb{K}[X]$ such that

- r is reduced w.r.t G;
- there is $g \in I$ s.t. $p = g + r$.

r is called the *normal form* of p w.r.t. G, and we denote it by $\bar{p}^G$.

For the proof of the above theorem we refer to (Cox *et al.*, 1998).

The interesting remaining part is to algorithmically formalize the computation of the Gröbner basis G of $I \trianglelefteq \mathbb{K}[X]$, from any basis $\{p_1, \ldots, p_s\}$ of I. For doing so, we introduce first the notion of *S-polynomials*, from which the *Buchberger Algorithm* for Gröbner basis construction is derived.

Definition 4.7 Let $p, q \in \mathbb{K}[X]$ be nonzero polynomials, and let $\prec$ be a monomial order on the set of terms of $X = \{x_1, \ldots, x_m\}$.

(1) If $\mathrm{LM}(p) = X^{\alpha}$ and $\mathrm{LM}(q) = X^{\beta}$, then let $\gamma = \{\gamma_1, \ldots, \gamma_m\}$, where $\gamma_i = \max(\alpha_i, \beta_i)$, $i = 1, \ldots, m$. The *least common multiple* of $\mathrm{LM}(p)$ and $\mathrm{LM}(q)$ is X^{γ}, and we write

$$X^{\gamma} = \mathrm{LCM}(\mathrm{LM}(p), \mathrm{LM}(q)).$$

(2) The *S-polynomial* of p and q, denoted by $S(p,q)$, is

$$S(p,q) = \frac{X^{\gamma}}{\mathrm{LT}(p)} p - \frac{X^{\gamma}}{\mathrm{LT}(q)} q.$$

Finally, the computation of a Gröbner basis of a given ideal $I \trianglelefteq \mathbb{K}[X]$ can be constructed in a finite number of steps by the following algorithm due to B. Buchberger.

Algorithm 4.8 (The Buchberger Algorithm for Gröbner Basis Computation (Buchberger, 2006))
Input: $P = \{p_1, \ldots, p_s\}$ s.t. $I = \langle p_1, \ldots, p_s \rangle \trianglelefteq \mathbb{K}[X]$, $I \neq \{0\}$
Output: Gröbner basis $G = \{g_1, \ldots, g_t\}$ for I w.r.t. $\prec$, with $P \subset G$
Assumption: $\prec$ is a fixed a monomial order on the set of terms of X

```
1  G:=P
2  repeat
3     G̃ := G
4     for each pair {u, v}, u ≠ v in G̃ do
5        S := overline(S(u, v))^G̃
6        if S ≠ 0 then G = G ∪ {S}
7  until G = G̃
8  return G.
```

Example 4.9 Let $P = \{p_1, p_2\}$, with $p_1 = x^2y^2 + y - 1$, $p_2 = x^2y + x \in \mathbb{Q}[x,y]$, and consider the *graduated lexicographic* (*glex*) ordering given below.

$$x^{\alpha_1} y^{\alpha_2} \prec x^{\beta_1} y^{\beta_2} \Leftrightarrow \alpha_1 + \alpha_2 < \beta_1 + \beta_2, \text{ with } x < y.$$

Thus $\mathrm{LT}(p_1) = x^2y^2, \mathrm{LT}(p_2) = x^2y$.

Using Algorithm 4.8, a Gröbner basis G of $\langle P \rangle \trianglelefteq \mathbb{Q}[x,y]$ is computed as follows.

1. $G = \{p_1, p_2\}$.

2. $S(p_1,p_2) = p_1 - yp_2 = -xy + y - 1 := p_3$.

p_3 is irreducible w.r.t. G, thus $G = \{p_1,p_2,p_3\}$.

3. $S(p_2,p_3) = p_2 + xp_3 = xy \underset{\text{reduced w.r.t.}p_3}{\longrightarrow} y - 1 := p_4$.

p_4 is irreducible w.r.t. G, thus $G = \{p_1,p_2,p_3,p_4\}$.

4. $S(p_3,p_4) = p_3 + xp_4 = y - x - 1 \underset{\text{reduced w.r.t.}p_4}{\longrightarrow} -x := p_5$.

p_5 is irreducible w.r.t. G, thus $G = \{p_1,p_2,p_3,p_4,p_5\}$.

5. All other S-polynomials reduce now to 0 w.r.t. G. Hence, the sought Gröbner basis is

$$G = \{x^2y^2 + y - 1,\ x^2y + x,\ -xy + y - 1,\ y - 1,\ -x\}.$$

Next, we can reduce the elements of G w.r.t. each other and normalize all leading coefficients to 1. This way, the obtained *reduced Gröbner basis* of $\langle p_1,p_2\rangle$ is

$$\{x,\ y - 1\}.$$

Using Algorithm 4.8, we can now state some ideal properties that are algorithmically decidable by Gröbner basis computation.

Theorem 4.10 Let $I = \langle p_1,\ldots,p_s\rangle \trianglelefteq \mathbb{K}[X]$, and let G be a Gröbner basis of I.

(1) *Ideal membership.* For a given polynomial $p \in \mathbb{K}[X]$ it can be decided whether $p \in I$ as follows.
$$p \in I \Leftrightarrow \bar{p}^G = 0.$$

(2) *Elimination.* A basis of the *elimination ideals*
$$I_j = I \cap \mathbb{K}[x_1,\ldots,x_j], \quad j = 1,\ldots,m,$$
can be computed.

(3) *Ideal equality.* Let $J = \langle q_1,\ldots,q_t\rangle \trianglelefteq \mathbb{K}[X]$, and let G_I and G_J be respectively the reduced Gröbner basis of I and J. Then
$$I = J \Leftrightarrow G_I = G_J.$$

(4) *Ideal sum, product and intersection.* Let $J = \langle q_1,\ldots,q_t\rangle \trianglelefteq \mathbb{K}[X]$. Then

- ideal sum:	$I + J = \langle p_1,\ldots,p_s,q_1,\ldots,q_t\rangle$;
- ideal product:	$I \cdot J = \langle p_i q_j \mid 1 \le i \le s, 1 \le j \le t\rangle$;
- ideal intersection:	$I \cap J = (\langle y\rangle \cdot I + \langle 1 - y\rangle \cdot J) \cap \mathbb{K}[X]$, where y is a new variable.

Based on ideal theoretic notions, as well as on the above mentioned properties of Gröbner bases theory, we have the following elimination property using ideals of polynomials over disjunct sets of variables.

Proposition 4.11 Let $I \trianglelefteq \mathbb{K}[X]$ and $J \trianglelefteq \mathbb{K}[Y]$, with $1 \notin I$, $1 \notin J$ and $X \cap Y = \emptyset$. Then

$$I + J \cap \mathbb{K}[X] = I. \tag{4.1}$$

Proof. Let $p \in I + J \trianglelefteq \mathbb{K}[X,Y]$, thus $p = a + b$, for $a \in I$ and $b \in J$.
Since $X \cap Y = \emptyset$, $I \trianglelefteq \mathbb{K}[X]$, $J \trianglelefteq \mathbb{K}[Y]$, if p is completely reduced w.r.t. I, then $a = 0$.
Hence, for showing (4.1) suffices to show

$$b \cap \mathbb{K}[X] = 0.$$

Since $1 \notin J$ and $b \in J \trianglelefteq \mathbb{K}[Y]$, b contains some variable of Y, and thus $b \cap \mathbb{K}[X] = 0$.
Hence $p \cap \mathbb{K}[X] \in I$, yielding

$$I + J \cap \mathbb{K}[X] = I.$$

■

Beside the already stated applications of polynomial ideal theory, Gröbner basis computation also offers the possibility of studying properties of polynomial maps defined as follows.

Definition 4.12 Given the polynomial rings $\mathbb{K}[X]$ and $\mathbb{K}[Y]$, where $X = \{x_1, \ldots, x_m\}$ and $Y = \{y_1, \ldots, y_n\}$, a *polynomial map* ϕ (or *polynomial map homomorphism*) is a ring homomorphism uniquely determined by

$$\phi : \mathbb{K}[x_1, \ldots, x_m] \to \mathbb{K}[y_1, \ldots, y_n], \qquad \phi : x_i \mapsto f_i, \tag{4.2}$$

where $f_i \in \mathbb{K}[y_1, \ldots, y_n]$, $i = 1, \ldots, m$.

The *kernel* of ϕ is the ideal

$$\ker\phi = \{g \in \mathbb{K}[X] \mid \phi(g) = 0\} \trianglelefteq \mathbb{K}[X]. \tag{4.3}$$

Given a polynomial $g \in \mathbb{K}[X]$, i.e. $g(X) = \sum_\alpha a_\alpha x_1^{\alpha_1} \cdots x_m^{\alpha_m}$, where $\alpha = \{\alpha_1, \ldots, \alpha_m\} \in \mathbb{N}^m$ and $a_\alpha \in \mathbb{K}$ with only finitely many non-zero a_α, then, by equation (4.2), we have

$$\phi(g) = \sum_\alpha a_\alpha f_1^{\alpha_1} \cdots f_m^{\alpha_m} \in \mathbb{K}[y_1 \ldots, y_n]. \tag{4.4}$$

Furthermore, we note that for any polynomial $g \in \mathbb{K}[X]$, we have

$$g \in \ker\phi \Leftrightarrow g(f_1, \ldots, f_m) = 0,$$

and therefore $\ker\phi$ is also called the *ideal of relations* among the polynomials $f_1, \ldots, f_m$.
Since $\ker\phi$ is an ideal, by Gröbner basis computation we can determine the generators for the kernel of a polynomial map homomorphism ϕ, based on the theorem below.

Theorem 4.13 (Adams and Loustaunau, 1994)
Let $A = \langle x_1 - f_1, \ldots, x_m - f_m \rangle \subset \mathbb{K}[x_1, \ldots, x_m, y_1, \ldots, y_n]$. Then

$$\ker \phi = A \cap \mathbb{K}[x_1, \ldots, x_m].$$

For the proof of the theorem we refer to (Adams and Loustaunau, 1994).

Hence, the algorithmic computation of a Gröbner basis for the kernel of ϕ can be synthesized as follows.

Algorithm 4.14 (Gröbner basis of $\ker\phi$**)**
Input: Polynomial map homomorphism ϕ as in (4.2)
Output: Gröber basis G of $\ker\phi$

1 By Algorithm 4.8, compute a Gröbner basis F for the ideal $A = \langle x_1 - f_1, \ldots, x_m - f_m \rangle$ with respect to an elimination order $\succ$ such that $y_1 \succ \cdots \succ y_n \succ x_1 \succ \cdots \succ x_m$

2 $G = F \cap \mathbb{K}[x_1, \ldots, x_m]$
3 **return** G.

Example 4.15 Consider a polynomial map $\phi : \mathbb{Q}[x,y,z] \to \mathbb{Q}[u,v]$ with

$$\begin{aligned} x &\mapsto u^4, \\ y &\mapsto u^2 * v^2, \\ z &\mapsto v^4. \end{aligned}$$

First we compute a Gröbner basis F for the ideal

$$A = \langle x - u^4, y - u^2 * v^2, z - v^4 \rangle \subset \mathbb{Q}[x,y,z,u,v]$$

with respect to an (glex) elimination order s.t. $u > v > x > y > z$. We get

$$F = \{-y^2 + x*z, v^4 - z, -v^2*y + u^2*z, -v^2*x + u^2*y, u^2*v^2 - y, u^4 - x\}.$$

Therefore, a Gröbner basis G for $\ker\phi$ is

$$G = F \cap \mathbb{Q}[x,y,z] = \{-y^2 + x*z\}.$$

4.2 Sequences and Recurrences

In this subsection we present several notions and properties from algorithmic combinatorics used further in the thesis.

Definition 4.16

- A univariate *sequence* in $\mathbb{K}$ is a function $f : \mathbb{N} \to \mathbb{K}$.
  By $f(n)$ we denote both the value of f at the point n and the whole sequence f itself.

- A sequence of the form $f(n) = \theta^n$, where $\theta \in \mathbb{K}$, is called an *exponential sequence* in $\mathbb{K}$.
- A *recurrence* for a sequence $f(n)$ is

$$f(n+r) = R(f(n), f(n+1), \ldots, f(n+r-1), n) \quad (n \in \mathbb{N}), \tag{4.5}$$

  for some function $R : \mathbb{K}^{r+1} \to \mathbb{K}$, where $r \in \mathbb{N}$ is the *order* of the recurrence.

The recurrence equation (4.5) of $f(n)$ allows the computation of $f(n)$ for any $n \in \mathbb{N}$. However, the recurrence gives us only an indirect way of computation of $f(n)$: computing $f(n)$ for an arbitrary n requires first to determine the previous values of the sequence $f(0),\ f(1), \ldots, f(n-1)$, and then, by (4.5), $f(n)$ is obtained. When n is a large number, this might take too long.

A *solution to the recurrence* would be thus more suitable for getting the value of $f(n)$ for any n. That is a *closed form* solution of $f(n)$ which does not require the computation of previous values $f(1), \ldots, f(n-1)$, and gives a faster and convenient way to determine the value of $f(n)$ for any n. In other words, the closed form solution of the recurrence is a function of the recurrence index n. The algorithmic computation of closed forms is called *symbolic summation*.

Since finding closed form expressions of recurrences in the general case is undecidable (see e.g. (Schneider, 2005)), it is necessary to distinguish among the type of recurrence equations. Moreover, it is needed to gain as much knowledge as possible about the already existing decision procedures for certain type of recurrences, and to apply them for computing the closed form solution of a given recurrence of a specific type.

In what follows, several classes of recurrences will be presented together with the algorithmic methods for solving them.

1 Gosper-summable Recurrences

Definition 4.17 A sequence $f(n)$ in $\mathbb{K}$ is called *hypergeometric* if there exists a rational function $t \in \mathbb{K}[x]$ such that

$$\frac{f(n+1)}{f(n)} = t(n),\ \forall n \in \mathbb{N}.$$

Example 4.18

- Polynomials $p \in \mathbb{K}[x]$ are hypergeometric.
- If $q \in \mathbb{K}$ then q is hypergeometric.
- Any sequence $f(n)$ defined by a product of factorials, binomials, pochhammers, rational-function terms and exponential expressions in the summation variable n (all these factors can be raised to an integer power) is hypergeometric.

Definition 4.19 A *Gosper-summable recurrence* f(n) in $\mathbb{K}$ is a recurrence

$$f(n+1) = f(n) + h(n+1) \quad (n \in \mathbb{N}), \tag{4.6}$$

where $h(n)$ is a hypergeometric sequence in $\mathbb{K}$.

The closed form solution of a Gosper-summable recurrence can be exactly computed using the decision algorithm given by R. W. Gosper (Gosper, 1978).

Algorithm 4.20 (The Gosper Algorithm for Closed Forms of Hypergeometric Sums (Gosper, 1978))
Input: A hypergeometric sequence $h(n)$
Output: A hypergeometric sequence $g(n)$ such that $g(n+1)-g(n)=h(n),\ \forall n\in\mathbb{N}$, or "impossible" if no such g exists

1 Compute $h_{rat}(x)$ such that $h_{rat}(n)=\dfrac{h(n+1)}{h(n)}$

2 Compute the *Gosper-form* of $h_{rat}(x)$ given by

$$h_{rat}(x)=\frac{p(x+1)}{p(x)}\frac{q(x)}{r(x+1)},\quad \text{where } p,\ q,\ r\in\mathbb{K}[x] \text{ and } gcd(q(x),r(x+j))=1,\ \forall j\in\mathbb{N}$$

3 Solve the Gosper-equation

$$p(x)=q(x)y(x+1)-r(x)y(x)\quad \text{for a polynomial solution}\quad y(x)\in\mathbb{K}[x]$$

4 If there is no such $y(x)$, **return** "impossible"

5 Compute $g_{rat}(x)=\dfrac{y(x+1)}{y(x)}\dfrac{q(x)}{r(x)}$

6 **return** $g(n)=\dfrac{1}{g_{rat}(n)-1}h(n)$.

The Gosper Algorithm solves the summation problem $\sum_{k=0}^{n-1}h(k)$ since it finds a hypergeometric sequence $g(n)$, if there exists one, satisfying the *telescoping equation* $g(n+1)-g(n)=h(n)$. This yields

$$g(n)-g(0)=\sum_{k=0}^{n-1}h(k).$$

Thus, having a Gosper-summable recurrence as in (4.6), we apply Algorithm 4.20 for its closed form computation.

Example 4.21 Given the Gosper-summable recurrence

$$f(n+1)=f(n)+n*2^n,\quad n\in\mathbb{N},$$

with initial value $f(0)$, find the closed form representation of $f(n)$.
The recurrence of $f(n)$ can be rewritten as

$$f(n+1)-f(n)=n*2^n,$$

and hence we have

$$f(n)=f(0)+\sum_{k=0}^{n-1}k*2^k.$$

For getting the closed form of $f(n)$ we thus need to compute the sum $\sum_{k=0}^{n-1}k*2^k$, where the summand $k*2^k$ is a hypergeometric term. Hence, we apply Algorithm 4.20, as follows.

- Denote $h(n) = n * 2^n$ and thus $h_{rat}(n) = 2 * \frac{n+1}{n}$.
- Using the Gosper-form of $h_{rat}(x)$, we take

$$p(x) = x,\ q(x) = 2,\ r(x) = 1.$$

- Solving the Gosper-equation, we have

$$x = 2 * y(x+1) - y(x),$$

  yielding that $y(x) = x - 2$.
- $g_{rat}(x) = 2 * \frac{x-1}{x-2}$.
- $g(n) = \frac{1}{2*\frac{n-1}{n-2}-1} * n * 2^n = 2^n * (n-2)$.

From the telescoping equation $g(n+1) - g(n) = h(n)$, we then have

$$\sum_{i=0}^{n-1} h(n) = g(n) - g(0) = 2^n(n-2) + 2.$$

Thus, the closed form of $f(n)$ is

$$f(n) = f(0) + 2 + 2^n(n-2).$$

Example 4.22 Using Algorithm 4.20, the closed forms of the following recurrence equations are presented below.

- $f(n+1) = f(n) + n^2 2^n \quad \underset{Gosper}{\Rightarrow} \quad f(n) = f(0) + 2^n(n^2 - 4n + 6) - 6.$
- $f(n+1) = f(n) + n * n! \quad \underset{Gosper}{\Rightarrow} \quad f(n) = f(0) - 1 + n!.$
- $f(n+1) = f(n) + n! \quad \underset{Gosper}{\Rightarrow}$ has no close form representation.

2 *C-finite Recurrences*

Definition 4.23 A *linear recurrence* in $\mathbb{K}$ is a recurrence

$$f(n+r) = a_0(n)f(n) + a_1(n)f(n+1) + \ldots + a_{r-1}(n)f(n+r-1) + a(n) \quad (n \in \mathbb{N}),$$

where $r \in \mathbb{N}$ is the *order* of the recurrence.
The recurrence is called *homogeneous* if $a(n) = 0$ and *inhomogeneous* otherwise.

A special case of linear recurrences are linear recurrences with constant coefficients, called *C-finite recurrences* (Zeilberger, 1990; Everest *et al.*, 2003).

Definition 4.24 A *C-finite recurrence* f(n) in $\mathbb{K}$ is a (homogeneous) linear recurrence with constant coefficients

$$f(n+r) = a_0 f(n) + a_1 f(n+1) + \ldots + a_{r-1} f(n+r-1) \quad (n \in \mathbb{N}), \tag{4.7}$$

where $r \in \mathbb{N}$ is the *order* of the recurrence, and $a_0, \ldots, a_{r-1}$ are constants from $\mathbb{K}$ with $a_0 \neq 0$.

By writing x^i for each $f(n+i),\ i = 0, \ldots, r$ in (4.7), the corresponding *characteristic polynomial* $c(x)$ of $f(n)$ is

$$c(x) = x^r - a_0 - a_1 x - \cdots - a_{r-1} x^{r-1}. \tag{4.8}$$

Example 4.25

(1) The following recurrences are C-finite.

- Fibonacci recurrence:

$$f(n+2) = f(n+1) + f(n), \quad f(0) = 0,\ f(1) = 1.$$

- Tribonacci recurrence:

$$f(n+3) = f(n+2) + f(n+1) + f(n), \quad f(0) = 0,\ f(1) = f(2) = 1.$$

(2) The recurrence

$$f(n+2) = (n+1) f(n+1) + n f(n), \quad f(0) = 1,$$

is not a C-finite recurrence, but it is a linear recurrence with polynomial coefficients in the summation variable n. Such recurrences are called *P-finite* recurrences.

(3) $f(n+1) = f(n)^2 + 5, \quad f(0) = 1$ is a non-linear recurrence.

Following (Zeilberger, 1990), we introduce an equivalent definition to Definition 4.24 of C-finite recurrences. For doing so, we first define the shift operator on univariate sequences.

Definition 4.26 The *shift operator S* is defined on univariate sequences $f : \mathbb{N} \to \mathbb{K}$ as

$$(S \cdot f)(n) = f(n+1) \quad (n \in \mathbb{N}). \tag{4.9}$$

Polynomials in $\mathbb{K}[S]$ represent linear constant coefficient recurrence operators. For example,

$$(S^2 - S) \cdot f = 0 \text{ is the recurrence } f(n+2) - f(n+1) = 0.$$

Definition 4.27 A sequence $f : \mathbb{N} \to \mathbb{K}$ is *C-finite* in $\mathbb{K}$ if it is annihilated by some nonzero operator $P \in \mathbb{K}[S]$, namely:

$$P \cdot f = 0, \quad P \in \mathbb{K}[S], \quad P \neq 0. \tag{4.10}$$

Definition 4.28 A system of recurrences for sequences $f_1(n), \ldots, f_m(n)$ is called *C-finite* (Gerhold, 2002; Kauers, 2005) if

$$\begin{aligned} f_i(n+r_i) \quad = \quad & a_{0,1} f_1(n) + a_{1,1} f_1(n+1) + \cdots + a_{r_i-1,1} f_1(n+r_i-1) \\ & + a_{0,2} f_2(n) + a_{1,2} f_2(n+1) + \cdots + a_{r_i-1,2} f_2(n+r_i-1) \\ & + \ldots \ \ldots \\ & + a_{0,m} f_m(n) + a_{1,m} f_m(n+1) + \cdots + a_{r_i-1,m} f_m(n+r_i-1) \quad (n \in \mathbb{N}), \end{aligned} \tag{4.11}$$

for some fixed orders $r_i \in \mathbb{N},\ i = 1, \ldots, m$, and constants $a_{i,j} \in \mathbb{K}$ with not all $a_{0,i}$ being zero.

A special case of a C-finite system is when all $f_i(n),\ i = 1,\ldots,m$, are C-finite sequences, whereas the general case presented in (4.11) is also called a *coupled C-finite system* (Zürcher, 1994; Gerhold, 2002).

A crucial and elementary fact about C-finite recurrences is that they always admit a closed form solution (Everest *et al.*, 2003).

Theorem 4.29 The closed form of a C-finite sequence $f(n)$ in $\mathbb{K}$ is

$$f(n) = p_1(n)\theta_1^n + \cdots + p_s(n)\theta_s^n, \tag{4.12}$$

where $\theta_1,\ldots,\theta_s \in \bar{\mathbb{K}}$ are the distinct roots of the characteristic polynomial of $f(n)$, and $p_i(n)$ are polynomials in n whose degree is less than the multiplicity of θ_i, $i = 1,\ldots,s$.

Proof. Let us take the general form of a C-finite recurrence

$$f(n+r) = a_0 f(n) + a_1 f(n+1) + \cdots + a_{r-1} f(n+r-1) \quad (n \in \mathbb{N}), \tag{4.13}$$

given by Definition 4.24

Since (4.13) is homogeneous, we immediately obtain the characteristic polynomial of $f(n)$ as

$$c(x) = x^r - a_0 - a_1 x - \cdots - a_{r-1}x^{r-1}.$$

Let $\theta_1, \theta_2, \ldots, \theta_s \in \bar{\mathbb{K}}$ be the distinct roots of the characteristic polynomial $c(x)$ and let e_i be the multiplicity of θ_i, $i = 1,\ldots,s$.

W.l.o.g., we may assume that $a_0 \neq 0$, thus we can assume that all $\theta_i \neq 0,\ i = 1,\ldots,s$.

Then, according to (Everest *et al.*, 2003), we have

$$f(n) = p_1(n)\theta_1^n + \cdots + p_s(n)\theta_s^n \tag{4.14}$$

for certain polynomials $p_i \in \mathbb{K}[n]$ of degree less than e_i, $i = 1,\ldots,s$.

The coefficients of the polynomials $p_i \in \mathbb{K}[n]$ are obtained by taking the first r initial values of the sequence $f(n)$.

■

The closed form of $f(n)$ given by (4.12) is called a *C-finite expression*. In other words a C-finite expression in n is a linear combination of exponential sequences θ^n in $\bar{\mathbb{K}}$ with polynomial coefficients in the summation variable n.

Using Theorem 4.29, the closed form computation of a C-finite recurrence in $\mathbb{K}$ is synthesized in the algorithm below.

Algorithm 4.30 (Closed Forms Computation of C-finite Recurrences (Zeilberger, 1990; Everest *et al.*, 2003))
Input: A C-finite recurrence equation $f(n+r) = a_{r-1}f(n+r-1) + \cdots + a_0 f(n)$ in $\mathbb{K}$, with r initial values
Output: Closed form solution of $f(n)$
Assumption: $a_i \in \mathbb{K}$, $i = 0,\ldots,r-1$, $r \geq 1$

1 Set up the characteristic polynomial of $f(n)$ as being

$$c(x) = x^r - a_{r-1}x^{r-1} - \cdots - a_1 x - a_0$$

2 Compute the complete factorization of $c(x)$ over $\bar{\mathbb{K}}$, namely, write

$$c(x) = (x-\theta_1)^{e_1} \cdots (x-\theta_s)^{e_s}, \quad e_i \in \mathbb{N},\ e_i \geq 1,\ i = 1,\ldots,s$$

3 The closed form $f(n)$ is the linear combination over $\bar{\mathbb{K}}$ of the sequences

$$\begin{cases} \theta_1^n, & n\theta_1^n, & \ldots, & n^{\underline{e_1-1}}\,\theta_1^n \\ \theta_2^n, & n\theta_2^n, & \ldots, & n^{\underline{e_2-1}}\,\theta_2^n \\ \ldots, & \ldots, & \ldots, & \ldots \\ \theta_s^n, & n\theta_s^n, & \ldots, & n^{\underline{e_s-1}}\,\theta_s^n \end{cases}, \text{ where } n^{\underline{a}} = n(n-1)\cdots(n-a+1) \text{ is the falling factorial of n}$$

4 By grouping the terms $n^d\theta_i^n$ in the closed form of $f(n)$, we have

$$f(n) = p_1(n)\theta_1^n + \ldots p_s(n)\theta_s^n, \quad \text{where } p_i(n) \in \bar{\mathbb{K}}[n]$$

(the coefficients of $p_i(n)$ are determined by the inital values of $f(n)$)

5 **return** $f(n) = p_1(n)\theta_1^n + \cdots + p_s(n)\theta_s^n$.

Based on Definition 4.17 and Theorem 4.29, terms $n^d\theta^n$ are both hypergeometric and C-finite expressions. However, there are infinitely many C-finite recurrences that are not Gosper-summable, and vice-versa. For example,

- the Fibonacci recurrence is C-finite but not Gosper-summable;
- the recurrence $f(n+1) = f(n) + n*n!$ is Gosper-summable but not C-finite.

Example 4.31 Given the C-finite recurrence

$$f(n+2) = 3f(n+1) - 2f(n) \quad (n \in \mathbb{N}),$$

with initial values $f(0) = 0,\ f(1) = 1$, find the closed form representation of $f(n)$.
We proceed similarly as in the proof of Theorem 4.29.

Step 1, 2. The characteristic polynomial of $f(n)$ is

$$x^2 - 3x + 2 = 0,$$

yielding the distinct roots $\theta_1 = 1$ and $\theta_2 = 2$, both roots with multiplicity 1.

Step 3, 4. The closed form of $f(n)$ is given by

$$f(n) = \alpha 1^n + \beta 2^n,$$

where the unknown constants α, β are determined using the initial values $f(0)$, $f(1)$.

We then have the system

$$\begin{cases} 0 = f(0) = \alpha+\beta \\ 1 = f(1) = \alpha+2\beta \end{cases},$$

yielding $\alpha = -1$ and $\beta = 1$.

Step 5. The closed form of $f(n)$ is thus

$$f(n) = 2^n - 1,\ \forall n \in \mathbb{N}.$$

An additional nice property of C-finite recurrences is that an inhomogeneous linear recurrence with constant coefficients, having as its inhomogeneous part a C-finite expression in n, can always be transformed into an equivalent (homogenous) C-finite recurrence. Thus, the recurrence

$$f(n+r) = a_0 f(n) + a_1 f(n+1) + \cdots + a_{r-1} f(n+r-1) + g(n) \quad (n \in \mathbb{N}),$$

where $a_0, \ldots, a_{r-1} \in \mathbb{K}$ and $g(n) \neq 0$ is a C-finite expression in n, always admits closed form solution. The closed form computation of such recurrences is presented below.

Proposition 4.32 Any first order inhomogeneous linear recurrence with constant coefficients given by

$$f(n+1) = af(n) + g(n) \quad (n \in \mathbb{N}), \tag{4.15}$$

where $a \in \mathbb{K}$ and $g(n) \neq 0$ is a C-finite expression in n, can be transformed into a homogeneous C-finite recurrence (with order greater or equal than 2).

Proof. We consider a first order inhomogeneous linear recurrence with constant coefficients as in (4.15).

$$f(n+1) = af(n) + g(n), \quad n \in \mathbb{N}, a \in \mathbb{K},\ g(n) \neq 0.$$

Based on Definition 4.27, we need to determine the operator P_f s.t. $P_f \cdot f = 0$, yielding the homogeneous C-finite recurrence of $f(n)$. For doing so, we proceed as follows.

(1) We determine the operator P_g s.t.

$$P_g \cdot g = 0, \tag{4.16}$$

and thus we have

$$P_g \cdot (f(n+1) - a * f(n)) = 0. \tag{4.17}$$

(2) Knowing that $(S-a) \cdot f = f(n+1) - af(n)$, we consider the operator

$$P_f = P_g \cdot (S-a). \tag{4.18}$$

We then obtain

$$P_f \cdot f = 0, \tag{4.19}$$

which gives us the homogeneous C-finite recurrence of $f(n)$.

Further, for obtaining the closed form of $f(n)$ we apply on (4.19) the general method using the characteristic polynomial presented in Theorem 4.29.

Thus, we need to determine the operator P_g for the C-finite expression $g(n)$ in n s.t. (4.16) holds. Based on (4.12), w.l.o.g., the C-finite expression $g(n)$ is

$$g(n) = n^{d_1}\theta_1^n + \cdots + n^{d_s}\theta_s^n,$$

where $\theta_1, \ldots, \theta_s$ are the (distinct) roots of the characteristic polynomial of $g(n)$ and d_i are at most the multiplicity e_i of θ_i, $i = 1, \ldots, s$.
Since $\theta_1, \ldots, \theta_s$ are the roots of the characteristic polynomial of $g(n)$, we have

$$(x-\theta_1)^{d_1+1}(x-\theta_2)^{d_2+1}\ldots(x-\theta_s)^{d_s+1} = 0, \tag{4.20}$$

which, by writing in (4.20) the shifting operator S instead of x, gives us the sought equation of P_g as presented below.

$$P_g = (S-\theta_1)^{d_1+1}(S-\theta_2)^{d_2+1}\ldots(S-\theta_s)^{d_s+1}, \text{ s.t. } P_g \cdot g = 0. \tag{4.21}$$

Thus

$$P_g \cdot (f(n+1) - af(n)) = 0,$$

that, using (4.18), yields P_f satisfying (4.19).

■

Definition 4.33 An inhomogeneous (or homogenous) linear recurrence with constant coefficients as in (4.15) is called an *affine relation*.

Corollary 4.34 Any inhomogeneous linear recurrence $f(n)$ of order $r \geq 1$ with constant coefficients given by

$$f(n+r) = a_0 f(n) + a_1 f(n+1) + \cdots + a_{r-1} f(n+r-1) + g(n) \quad (n \in \mathbb{N}), \tag{4.22}$$

where $a_0, \ldots, a_{r-1} \in \mathbb{K}$ and $g(n) \neq 0$ is a C-finite expression in n, can be transformed into a homogeneous C-finite recurrence (with order greater or equal than $r+1$).

Proof. We proceed in a similar way as for Proposition 4.32.

(1) Determine P_g s.t. $P_g \cdot g = 0$.

(2) Consider the operator

$$P_f = P_g \cdot (S^r - a_{r-1}S^{r-1} - \cdots - a_1 S - a_0) \text{ s.t. } P_f \cdot f = 0,$$

yielding the homogenous C-finite recurrence of $f(n)$.

■

Hence, the algorithm for closed form computation of an inhomogeneous linear recurrence with constant coefficients, having as its inhomogeneous part a C-finite expression, is as follows.

Algorithm 4.35 (Closed Form Computation of Inhomogeneous C-finite Recurrences)
Input: Inhomogeneous linear recurrence with constant coefficients and C-finite inhomogeneous part, defined as

$$f(n+r) = a_0 f(n) + a_1 f(n+1) + \cdots + a_{r-1} f(n+r-1) + n^{d_1}\theta_1^n + \cdots + n^{d_s}\theta_s^n \quad (n \in \mathbb{N})$$

Output: Closed form solution of $f(n)$
Assumption: $a_i \in \mathbb{K}$, $i = 0,\ldots,r-1$, $r \geq 1$, $\theta_j \in \bar{\mathbb{K}}$, $d_j \in \mathbb{N}$, $j = 1,\ldots,s$

1 Rewrite the recurrence as

$$f(n+r) - a_0 f(n) - a_1 f(n+1) - \cdots - a_{r-1} f(n+r-1) = n^{d_1}\theta_1^n + \cdots + n^{d_s}\theta_s^n$$

2 From the inhomogeneous C-finite part construct the operator

$$P_g = (S-\theta_1)^{d_1+1}(S-\theta_2)^{d_2+1}\cdots(S-\theta_s)^{d_s+1}$$

3 Take $P_f = P_g \cdot (S^r - a_{r-1}S^{r-1} - \cdots - a_1 S - a_0)$, and denote $\widetilde{f} := P_f \cdot f$
4 Apply Algorithm 4.30 for computing the closed form of $\widetilde{f}(n)$
5 **return** the closed form of $\widetilde{f}(n)$, i.e. of $f(n)$.

Example 4.36 The equivalent (homogenous) C-finite recurrence of the following inhomogeneous linear recurrences with constant coefficients are listed below.

- $f(n+2) = 2f(n) + 3f(n) + n$

  $\underset{P_f=(S^2-2S-3)(S-1)^2}{\Rightarrow}$ $f(n+4) - 4f(n+3) + 2f(n+2) + 4f(n+1) - 3f(n) = 0.$

- $f(n+1) = f(n) + n^2 * 2^n$

  $\underset{P_f=(S-1)(S-2)^3}{\Rightarrow}$ $f(n+4) - 7f(n+3) + 18f(n+2) - 20f(n+1) + 8f(n) = 0.$

3 Generating Functions

Definition 4.37 Given an infinite sequence $\{f(0), f(1), f(2), \ldots\}$, its *generating function* (Graham *et al.*, 1989) is the *power series* representation

$$F(z) = f(0) + f(1)z + f(2)z^2 + \cdots = \sum_{n\in\mathbb{N}} f(n)z^n$$

in an auxiliary variable z.

Informally, the generating function $F(z)$ of the infinite sequence $f(n)$ generates the coefficient of interest, namely, the elements of the sequence $f(n)$.

The "advantage" of using generating functions is that we can represent and reason about the entire infinite sequence by the single quantity representing its generating function. Often, one

can solve problems involving sequences by first setting up one or more generating functions, then gain necessary knowledge by playing around with these generating functions, and finally, looking again at the coefficients of the generating functions, the needed information about the sequences are obtained.

The algorithmic usage of generating functions for solving recurrences is presented below.

Algorithm 4.38 (Solving Recurrences by the Technique of Generating Functions (Graham *et al.*, 1989))
Input: The recurrence equation of $f(n),\ n \in \mathbb{N}$
Output: Closed form for $f(n)$
Assumption: $f(n) = 0, \forall n < 0$

1 Consider the recurrence equation of $f(n)$, valid for all integers $n \in \mathbb{Z}$
2 Multiply both sides of the equation by z^n and sum over all n. The left hand side thus yields

$$F(z) = \sum_n f(n) z^n$$

3 The right hand side is manipulated s.t. it becomes an expression involving $F(z)$
4 Solve the resulting equation and thus determine the closed form of $F(z)$
5 Expand $F(z)$ into a power series and read off the coefficient of z^n, that is

$$\text{a closed form for } f(n)$$

6 **return** the closed form of $f(n)$.

We introduce the notation

$$[z^n]R(z),$$

for denoting the coefficient of z^n in $R(z)$.

The "tricky" part of Algorithm 4.38 is Step 5, where we are interested in the coefficients of z^n in the power series expansion of the obtained closed form of the generating function. Thus, we need to "guess" the appropriate power series expansion. Fortunately, there is an algorithmic computation of power series expansion for rational functions $R(z) = \frac{P(z)}{Q(z)}$, where P and Q are polynomials, given by the next theorem.

Theorem 4.39 Rational Expansion Theorem (Graham *et al.*, 1989)
If $R(z) = \frac{P(z)}{Q(z)}$, where $Q(z) = q_0(1-\rho_1 z)\cdots(1-\rho_s z)$, the numbers $\rho_1, \ldots, \rho_s$ are distinct, and $P(z)$ is a polynomial of degree less than s, then

$$[z^n]R(z) = a_1\rho_1^n + \cdots + a_s\rho_s^n, \quad \text{where } a_i = \frac{-\rho_i P(1/\rho_i)}{Q'(1/\rho_i)},\ i = 1, \ldots, s. \tag{4.23}$$

Note that every rational function $R(z)$ can be written as $R(z) = \frac{P(z)}{Q(z)} + T(z)$, where the degree of P is less than the degree of Q, and T is a polynomial. Further, by Theorem 4.39, the closed form of the coefficient $[z^n]\frac{P(z)}{Q(z)}$ is determined. Since $T(z)$ is a polynomial in z, a closed form for the coefficient $[z^n]R(z)$ is thus computed.

Example 4.40 Given the recurrence

$$f(0)=0,\ f(1)=1, f(n)=f(n-1)+f(n-2), \quad n\geq 2,$$

its closed form solution by Algorithm 4.38 is obtained as follows.

Step 1. The "single" recurrence equation of $f(n)$, without case-distinctions on the values of $n \in \mathbb{Z}$ ($f(n)=0,\ n<0$), is

$$f(n)=f(n-1)+f(n-2)+[n=1],$$

where $[n=1]$ adds 1 (i.e. the value of $f(1)$) to the right hand side of the recurrence equation when $n=1$, and makes no change when $n\neq 1$.

Steps 2, 3. Multiplying by z^n and summing over all $n\in\mathbb{Z}$, we have

$$\begin{aligned} F(z) &= \textstyle\sum_n f(n)z^n &&= \textstyle\sum_n f(n-1)z^n + \sum_n f(n-2)z^n + \sum_n [n=1]z^n \\ &&&= zF(z)+z^2F(z)+z. \end{aligned}$$

Step 4. We get

$$F(z)=\frac{z}{1-z-z^2}.$$

Step 5. The roots of $z^2+z-1=0$ are $\rho_1=\frac{1+\sqrt{5}}{2}$ and $\rho_2=\frac{1-\sqrt{5}}{2}$. By Theorem 4.39 we thus obtain the closed form of $f(n)$ as given below.

$$f(n) = [z^n]F(z) = \frac{\rho_1^n-\rho_2^n}{\sqrt{5}}.$$

Theorem 4.39 is applicable for the case when the roots of $Q(z)$ are distinct. Fortunately, there is also an algorithmic "way-out" for getting the coefficients of z^n in $R(z)=\frac{P(z)}{Q(z)}$ in the case when $Q(z)$ has repeated roots, as presented below.

Theorem 4.41 General Expansion Theorem for Rational Generating Functions (Graham *et al.*, 1989)
If $R(z)=\frac{P(z)}{Q(z)}$, where $Q(z)=q_0(1-\rho_1 z)^{d_1}\dots(1-\rho_s z)^{d_s}$, the numbers $\rho_1,\dots,\rho_s$ are distinct, and $P(z)$ is a polynomial of degree less than $d_1+\dots+d_s$, then

$$[z^n]R(z)=f_1(n)\rho_1^n+\dots+f_s(n)\rho_s^n, \qquad \text{for all } n\geq 0, \tag{4.24}$$

where each $f_i(n)$, $i=1,\dots,s$, is a polynomial of degree d_i-1 with leading coefficient

$$\begin{aligned} a_i &= \frac{(-\rho_i)^{d_i}P(1/\rho_i)d_i}{Q^{(d_i)}(1/\rho_i)} \\ &= \frac{P(1/\rho_i)}{(d_i-1)!q_0\prod_{j\neq i}(1-\rho_j/\rho_i)^{d_j}}\ . \end{aligned} \tag{4.25}$$

4.3 Algebraic Dependencies of Exponential Sequences

Combining the theory of polynomial ideals with the theory of sequences and recurrences, in this subsection we present basic definitions, properties and results of polynomial relations among exponential sequences.

Definition 4.42 Let $\theta_1,\ldots,\theta_s \in \bar{\mathbb{K}}$ be algebraic numbers, and their corresponding exponential sequences $\theta_1^n,\ldots,\theta_s^n$, $n \in \mathbb{N}$.

An *algebraic dependency* (or *algebraic relation*) of these sequences over $\bar{\mathbb{K}}$ is a polynomial $p \in \bar{\mathbb{K}}[x_1,\ldots,x_s]$ in s distinct variables $x_1,\ldots,x_s$, i.e. in as many distinct variables as exponential sequences, such that p vanishes when variables are substituted by the exponential sequences, namely:

$$p(\theta_1^n,\ldots,\theta_s^n) = 0, \quad \forall n \in \mathbb{N}. \tag{4.26}$$

For (4.26), we also say that $\theta_1^n,\ldots,\theta_s^n$ are *polynomially related*.

Thus, computation of *all* algebraic dependencies among the algebraic exponential sequences $\theta_1^n,\ldots,\theta_s^n$ reduces to determine the *ideal of polynomials relations* $I(\theta_1^n,\ldots,\theta_s^n)$ among them. Hence, a basis for this ideal is needed from which any algebraic dependency can be derived.

Theorem 4.43 (Kauers and Zimmermann, 2006)

The algebraic relations among the algebraic exponential sequences $\theta_1^n,\ldots,\theta_s^n$ are generated by the algebraic relations among $n,\theta_1^n,\ldots,\theta_s^n$.

We thus have

$$I(\theta_1^n,\ldots,\theta_s^n) = I(n,\theta_1^n,\ldots,\theta_s^n) \trianglelefteq \bar{\mathbb{K}}[x_0,x_1,\ldots,x_s], \tag{4.27}$$

where the variables $x_0,x_1,\ldots,x_s$ stand for the C-finite sequences $n,\theta_1^n,\ldots,\theta_s^n$.

In other words, the linear sequence $n \mapsto n$ does not introduce any new algebraic dependency among $\theta_1^n,\ldots,\theta_s^n$.

For the proof we refer to (Kauers and Zimmermann, 2006).

Example 4.44

(1) The algebraic dependency among the exponential sequences of $\theta_1 = 2$ and $\theta_2 = 4$ over $\bar{\mathbb{Q}}$ is

$$\theta_1^{2n} - \theta_2^n = 0.$$

(2) The algebraic dependency among the exponential sequences of $\theta_1 = 2$ and $\theta_2 = \frac{1}{2}$ over $\bar{\mathbb{Q}}$ is

$$\theta_1^n * \theta_2^n - 1 = 0.$$

(3) The algebraic dependency among the exponential sequences of $\theta_1 = \frac{1+\sqrt{5}}{2}, \theta_2 = \frac{1-\sqrt{5}}{2}$ over $\bar{\mathbb{Q}}$ is

$$\left(\theta_1^n\right)^2 * \left(\theta_2^n\right)^2 - 1 = 0.$$

(4) There is no algebraic dependency among the exponential sequences of $\theta_1 = 2$ and $\theta_2 = 3$ over $\bar{\mathbb{Q}}$.

Note that the multiplicative relations among $\theta_1,\ldots,\theta_s$ imply corresponding relations among $\theta_1^n,\ldots,\theta_s^n$. Based on Theorem 4.43, the algebraic relations of $n,\theta_1^n,\ldots,\theta_s^n$ can thus be algorithmically computed by the following algorithm.

Algorithm 4.45 (Algebraic Dependencies of Algebraic Exponential Sequences (Ge, 1993; Kauers and Zimmermann, 2006))
Input: $\theta_1,\ldots,\theta_s \in \bar{\mathbb{K}}$, $n \in \mathbb{N}$
Output: A set $\{g_1,\ldots,g_t\} \subset \bar{\mathbb{K}}[x_0,x_1,\ldots,x_s]$ such that $I(n,\theta_1^n,\ldots,\theta_s^n) = \langle g_1,\ldots,g_t\rangle$

1 Determine the set of all integer tuples

$$L = \{(m_1,\ldots,m_s) \in \mathbb{Z}^s \mid \prod_{i=1}^{s} \theta_i^{m_i} = 1\}$$

2 Introduce new variables $x_0,\ldots,x_s$ for denoting

$$x_0 := n,\ x_1 := \theta_1^n,\ \ldots, x_s := \theta_s^n$$

3 By Algorithm 4.8, compute a basis $G = \{g_1,\ldots,g_t\}$ of the ideal

$$\Big\langle \Big\{ \prod_{i=1}^{s} x_i^{a_i} - \prod_{i=1}^{s} x_i^{b_i} \mid a \in \mathbb{N}^s, b \in \mathbb{N}^s, a-b \in L \Big\} \Big\rangle$$

4 **return** G.

Step 1 is performed using the results of (Ge, 1993; Borwein and Lisoněk, 2000) by computing a finite set $\{v_1,\ldots,v_r\} \subset \mathbb{Z}^s$ s.t. $L = \mathbb{Z}v_1 + \ldots \mathbb{Z}v_r$.

Example 4.46 Using Algorithm 4.45, the algebraic dependencies among θ_1^n and θ_2^n over $\bar{\mathbb{Q}}$, where $\theta_1 = \frac{1+\sqrt{5}}{2}$ and $\theta_1 = \frac{1-\sqrt{5}}{2}$, are obtained as follows.

Step 1. $L = (2,2)\cdot\mathbb{Z}$ corresponding to $\theta_1^2\theta_2^2 = 1$.

Step 2. Denote $x_0 := n,\ x_1 := \theta_1^n,\ x_2 := \theta_2^n$.

Step 3. Take $a := (2,2) \in \mathbb{N}^2,\ b := (0,0) \in \mathbb{N}^2$. Thus $a-b \in L$, and we have

$$G = \{x_1^2x_2^2 - 1\}.$$

Hence the algebraic dependency among θ_1^n and θ_2^n is

$$\left(\theta_1^n\right)^2\left(\theta_2^n\right)^2 - 1 = 0.$$

Using results of (Kauers and Zimmermann, 2006), based on Algorithm 4.45, we can also compute algorithmically the algebraic dependencies of *a system of algebraic exponential sequences*, as it is formulated below.

Theorem 4.47 (Kauers and Zimmermann, 2006)

Given the sequences $f_{i,j} : \mathbb{N} \to \bar{\mathbb{K}}$, where d, $s \in \mathbb{N}$, $i = 1, \ldots, d$, $j = 0, \ldots, s$, with $f_{i,0}(n_i) = n_i$, $f_{i,k}(n_i) = \theta_{i,k}^{n_i}$, $k = 1, \ldots, s$, $\theta_{i,k} \in \bar{\mathbb{K}}$. In the system of (C-finite) sequences

$$\begin{array}{ccc} f_{1,0}(n_1), & \ldots, & f_{1,s}(n_1), \\ f_{2,0}(n_2), & \ldots, & f_{2,s}(n_2), \\ \ldots, & \ldots, & \ldots, \\ f_{d,0}(n_d), & \ldots, & f_{d,s}(n_d) \end{array}$$

the sequences from the i^{th} row depend thus *only* on n_i. Let $I_i = I(f_{i,0}, f_{i,1}, \ldots, f_{i,s})$ be the ideal of algebraic dependencies of the sequences from the ith row.

Then the ideal I_* of algebraic dependencies among $f_{1,0}, \ldots, f_{1,s}, \ldots \ldots, f_{d,0}, \ldots, f_{d,s}$ is determined by the sum of the ideals I_i, and we have

$$I_* = \sum_{i=1}^{d} I_i. \tag{4.28}$$

For the proof we refer to (Kauers and Zimmermann, 2006).

Theorem 4.47 handles a special case of C-finite sequences. Namely, it determines the algebraic dependencies only among algebraic exponential sequences in the recurrence variables n_i and the linear sequences $n_i \mapsto n_i$.

However, it is worth to be mentioned that (Kauers and Zimmermann, 2006) can handle the general case of systems of arbitrary C-finite sequences (in other words, C-finite multisequences).

5 P-solvable Loops with Assignment Statements Only

Programs (with loops) usually can be elegantly expressed by means of recurrence relations, and thus the wide theory of recurrence equations offers a smooth way to assist programmers in reasoning about the behavior of a program.

The values of the program variables at compile time are not available; a technique is required to statically discover properties of programs. This can be achieved by specifying the relationship between the actual values treated symbolically and their description, in other words, by expressing the values of variables in a current state in terms of their values in previous states. This means, the behavior of variables are characterized by recurrence equations. Hence, finding exact solutions as closed forms in an acceptable timescale for a large class of recurrence equations is a challenging task in reasoning about imperative loops, for automatically inferring polynomial invariant relations among the loop variables.

In other words, a recurrence equation of a loop variable allows the "indirect" computation of the loop variable's value at any loop iteration, by first determining the values of the loop variable from previous loop iterations, and then applying the recurrence equation to get its current values. When the number of iterations is large, this might take too long. Thus, a *closed form solution to the recurrence* would be more suitable for getting the value of the loop variable for any loop iteration without computing its previous values.

Example 5.1 Consider Example 2.2. Denoting by $n \geq 0$ the loop counter, the behavior of the program

$$\begin{array}{l} quo := 0;\ rem := x; \\ \texttt{While}[y \leq rem, \\ \qquad rem := rem - y; \\ \qquad quo := quo + 1] \end{array}$$

is described by the system of recurrence equations of the loop variables given below.

$$\begin{cases} quo[n+1] = & quo[n] + 1 \\ rem[n+1] = & rem[n] - y \end{cases},$$

with the initial values defined as follows:

$$quo[0] = 0,\ rem[0] = x.$$

Solving this system of recurrences would thus describe the behavior of the loop variables in a "direct" way, i.e. in terms of the loop counter and the initial values of the loop variables.

Techniques for automatically checking and finding loop invariants and intermediate assertions have been studied and developed since the early works of German and Wegbreit (1975); Karr (1976); Cousot and Cousot (1977b, 1979). Recently, due to the increased computing power of hardware, as well as advances in methods for symbolic computation and automated theorem proving (Buchberger, 1996; Robinson and Voronkov, 2001), the problem of automated invariant generation is once again getting considerable attention. Particularly, using the abstract interpretation framework (Cousot and Halbwachs, 1978), many researchers (Müller-Olm and Seidl, 2004b,c; Rodriguez-Carbonell and Kapur, 2007a; Sankaranaryanan *et al.*, 2004; Kapur, 2006) have proposed methods for automatically computing *polynomial invariant* identities using polynomial ideal theoretic algorithms (Buchberger, 2006, 1985).

Following the "trend" of developing powerful algorithms for automatically inferring polynomial invariants, in our work we do not use the framework of abstract interpretation, but apply advanced methods from symbolic summation (Zeilberger, 1990; Paule and Schorn, 1995; Mallinger, 1996; Kauers and Zimmermann, 2006) together with polynomial algebra algorithms (Buchberger, 2006; Winkler, 1996; Buchberger, 1998).

Based on the shape of the loop body and on the structure of assignment statements, we define a certain family of loops, called *P-solvable*, for which the value of each program variable can be expressed as a polynomial of the initial values of variables, the loop counter, and some new variables where there are algebraic dependencies among the new variables.

If the bodies of these loops consist only of assignments and conditional branches, and test conditions in the loop and conditionals are ignored, we present a systematic method for generating polynomial invariants. Moreover, the approach is shown to be complete for some special cases. By completeness we mean that it generates a set of polynomials from which, under additional assumptions for loops with conditional branches, any polynomial invariant can be derived.

As already mentioned, in our approach for generating polynomial invariants, test conditions in the loops and conditionals are ignored. This turns the considered loops into *non-deterministic* program fragments. In what follows, the syntax and semantics of such programs is first presented, together with relevant notations used further in the thesis.

5.1 Basic Non-deterministic Programs

When we ignore conditions of loops, we will essentially deal with non-deterministic programs. Let us introduce a regular expression-like notation for non-deterministic programs and their semantics. Semantically, we consider a non-deterministic program S as the set of its possible execution paths. Each of these paths will be a path in some deterministic program S', so we can also regard a non-deterministic program as a collection of such deterministic programs.

We will only consider non-deterministic programs of a special form, these programs will be called *basic non-deterministic program*. To define the semantics of these programs we introduce a binary relation $S \sqsubseteq P$, where S is a sequence of assignments and P a basic non-deterministic program, having in mind that $S \sqsubseteq P$ implies that S is a possible sequence of assignments performed by P.

Definition 5.2 The notion of *basic non-deterministic program* is defined as follows.

(1) If S is a sequence of assignments, then S is also a basic non-deterministic program.

(2) If $P_1,\ldots,P_k$ are basic non-deterministic program, then $P_1|\ldots|P_k$, $P_1;\ldots;P_k$, and $P_1*\ldots*P_k$ are basic non-deterministic programs.

(3) If P is a basic non-deterministic program and j a non-negative integer, then P^j is a basic non-deterministic program.

(4) If P is basic non-deterministic program, then P^* is a basic non-deterministic program too.

The semantics of basic non-deterministic programs is defined using a relation $\sqsubseteq$ as follows.

Definition 5.3 Let us define a binary relation $\sqsubseteq$ between sequences of assignments and basic non-deterministic programs as follows. In this definition S denotes a sequence of assignments, k a non-negative integer and $P,P_1,\ldots,P_k$ basic non-deterministic programs.

(1) If P is a sequence of assignments, then $S\sqsubseteq P$ if and only if S coincides with P.

(2) We have $S\sqsubseteq(P_1|\ldots|P_k)$ if and only if for some $i=1,\ldots,k$ we have $S\sqsubseteq P_i$.

(3) We have $S\sqsubseteq(P_1;\ldots;P_k)$ if and only if S has the form $S_1;\ldots;S_k$, where $S_1,\ldots,S_k$ are sequences of assignments, and for each $i=1,\ldots,k$ we have $S_i\sqsubseteq P_i$.

(4) We have $S\sqsubseteq(P_1*\ldots*P_k)$ if and only if for some permutation W of $\{1,\ldots,k\}$ we have $S\sqsubseteq P_{W_1};\ldots;P_{W_k}$.

(5) We have $S\sqsubseteq P^k$ if and only if S has the form $S_1;\ldots;S_k$, where $S_1,\ldots,S_k$ are sequences of assignments, and for each $i=1,\ldots,k$ we have $S_i\sqsubseteq P$.

(6) We have $S\sqsubseteq P^*$ if and only if $S\sqsubseteq P^k$ for some non-negative integer k.

For a basic non-deterministic programs P we will write $\{Q\}P\{R\}$ to mean that $\{Q\}S\{R\}$ for all sequences of assignments $S\sqsubseteq P$. For example, it is not hard to argue that $\{Q\}P_1|P_2\{R\}$ if and only if $\{Q\}P_1\{R\}$ and $\{Q\}P_2\{R\}$.

For two basic non-deterministic programs P_1,P_2 we write $P_1\sqsubseteq P_2$ if for every sequence of assignments S, if $S\sqsubseteq P_1$, then $S\sqsubseteq P_2$. We call basic non-deterministic programs P_1 and P_2 *equivalent* if $P_1\sqsubseteq P_2$ and $P_2\sqsubseteq P_1$. Essentially, equivalence means that any sequence of assignments that can be performed by P_1 can also be performed by P_2 and vice versa. We have the following obvious corollary.

Corollary 5.4 Let P_1,P_2 be basic non-deterministic programs and Q,R assertions.

(1) If $P_2\sqsubseteq P_1$ and $\{Q\}P_1\{R\}$, then $\{Q\}P_2\{R\}$.

(2) If P_1 and P_2 are equivalent, then $\{Q\}P_1\{R\}$ if and only if $\{Q\}P_2\{R\}$.

In the sequel we will call basic non-deterministic programs simply programs.

Now we can formalize what we mean by conditional statements and while loops with ignored conditions. When we omit the condition b from a conditional statement

$$\texttt{If}[b\ \texttt{Then}\ S_1\ \texttt{Else}\ S_2]$$

we will write it in the form

$$\texttt{If}[\ldots\ \texttt{Then}\ S_1\ \texttt{Else}\ S_2]$$

and mean the basic non-deterministic program $S_1|S_2$. Likewise, when we omit the condition b from a loop

$$\texttt{While}[b,S]$$

we will write it in the form

$$\texttt{While}[\ldots,S]$$

and mean the basic non-deterministic program S^*.

5.2 Polynomial Invariants and Ideals

Let us first fix some notation.

We denote by $X = \{x_1,\ldots,x_m\},\ m > 1$, the set of (recursively changed) loop variables having $\mathbb{K}$ as the domain of their values. The set of initial values of the loop variables from $\mathbb{K}$ (i.e. before entering the loop) is denoted by $X_0 = \{x_{01},\ldots,x_{0m}\} \in \mathbb{K}^m$, where $x_{0i} \in \mathbb{K}$ is the initial value of $x_i,\ i = 1,\ldots,m$. For simplicity, we write $X = X_0$ for the sequence of assignments $x_i = x_{0i}$ assigning each variable its initial value.

Throughout this section, $n \in \mathbb{N}$ is a non-negative integer denoting the loop counter. Following notations from Sections 2, we consider while loops as presented below.

$$\texttt{While}[b,\ S], \tag{5.1}$$

where

- b is the boolean test condition of the loop. However, as presented in Subsection 5.1, in our approach for generating polynomial invariants, *test conditions in the loops are ignored*;
- S is a sequence of assignment statements in the loop variables X. In other words, S is the body of the loop consisting of assignment statements only. For any non-negative integer $n \in \mathbb{N}$, we denote by S^n the sequence $\underbrace{S;\ldots;S}_{n \text{ times}}$ for referring to the n times repeated execution the loop body S.

  Since loop conditions are ignored, based on Subsection 5.1, instead of S^n we also write S^* to refer to the execution of S arbitrary many times.

Using Hoare triple notation, let us now define the notion of *polynomial invariants among the loop variables X with initial values X_0 for P-solvable loops with assignments only as in* (5.1).

Definition 5.5 A polynomial $p \in \mathbb{K}[X]$ is a *polynomial invariant* of (5.1) among the loop variables X with initial values X_0 if for every non-negative integer n we have

$$\{p(X) = 0 \wedge X = X_0\} \quad S^n \quad \{p(X) = 0\}. \tag{5.2}$$

If (5.2) is valid, we also say that p is a *polynomial invariant among the loop variables X parameterized by X_0*.

Note that when S is a basic non-deterministic program, instead of writing that (5.2) holds for all n we can simple write

$$\{p(X) = 0 \wedge X = X_0\} \quad S^* \quad \{p(X) = 0\}.$$

In other words, $p \in \mathbb{K}[X]$ is a polynomial invariant for (5.1) among the loop variables X with initial values X_0 if and only if the following conditions are fulfilled.

(1) $p(X)$ is valid for the initial values of the loop variables. Namely, $p(X)$ is 0 whenever X are evaluated at X_0. Thus, $p(X)$ *holds at the entry/beginning of the loop*.

(2) $p(X)$ is valid after arbitrary many execution of the loop body S starting from a state (precondition) fulfilling the initial values X_0 of the loop variables X. Thus, $p(X)$ is *preserved by any execution of the loop body*.

Remark 5.6 Using the weakest precondition method, the validity of the Hoare triple (5.2) reduces to proving the implication

$$p(X) = 0 \wedge X = X_0 \implies \mathrm{wp}(S^n, p(X) = 0). \tag{5.3}$$

Since S is a sequence of assignments, calculation of $\mathrm{wp}(S^n, p(X) = 0)$ is straightforward by applying the wp-rules for sequencing and assignments, yielding a polynomial equation among the loop variables X. More precisely, $\mathrm{wp}(S^n, p(X) = 0)$ returns a polynomial equation among X by suitably modifying all loop variables in $p(X) = 0$ which possibly change due to the assignments $\underbrace{S;\ldots;S}_{n \text{ times}}$. Proving formula (5.3) is thus an ideal membership problem. Namely, it reduces to proving that the polynomial relation corresponding to the polynomial equation $\mathrm{wp}(S^n, p(X) = 0)$ is in the ideal generated by $\{p(X), X - X_0\}$.

Note that Definition 5.5 defines polynomial invariants for *loops with assignments only*. However, this is extended by Definition 6.11 in Section 6 for *loops with conditional branches and assignments*.

As it was observed already by Rodriguez-Carbonell and Kapur (2004); Kapur (2006), the set of polynomial invariants forms a polynomial ideal, called *polynomial invariant ideal*, in the following sense.

Theorem 5.7 The set of polynomial invariants among the loop variables $X = \{x_1, \ldots, x_m\}$ with initial values X_0 form a polynomial ideal in $\mathbb{K}[X]$.

By Hilbert's basis theorem (Adams and Loustaunau, 1994), any ideal, and in particular thus the ideal of polynomial invariants, has a finite basis. Using the Buchberger Algorithm (see Algorithm 4.8 and Buchberger (2006)), a Gröbner basis $\{p_1, \ldots, p_r\}$ of the polynomial invariant ideal can be effectively computed. Hence, the conjunction of the polynomial equations corresponding to the polynomials from the computed basis (i.e. $p_i(X) = 0$) characterizes completely the invariants of the loop. Namely, any other invariant can be derived from the computed basis $\{p_1, \ldots, p_r\}$.

The challenging task is thus to determine the *ideal of polynomial invariants*. The rest of the thesis addresses this issue.

Definition 5.8 If $p(X)$ is a polynomial invariant of loop (5.1), then we say that $p(X) = 0$ is a *polynomial invariant equation* of this loop. An assertion I is called a *polynomial (or algebraic) loop invariant* of (5.1) if I is a conjunction of polynomial invariant equations of this loop.

Polynomial relations as invariants found by an automatic analysis play a significant role in program verification, since they provide non-trivial valid assertions about the program. Polynomial invariants express complex relationships among program variables, the discovered assertions express essential properties of the program, and thus significantly simplify the verification task.

Based on the shape of the loop body and on the structure of assignment statements, our algorithm for polynomial invariant generation combines computer algebra and algorithmic combinatorics, in such a way that at the end of the invariant generation process polynomial invariants of a *P-solvable loop* (see Definitions 5.9 and 6.8) are automatically obtained.

In the case of P-solvable loops with assignments only, we show in Subsection 5.4 that our method is complete. Namely, it finds a basis for the polynomial invariant ideal of the loop. The approach is illustrated on many examples in Subsection 5.8.

Further, for P-solvable loops with conditional branches and assignments, under the additional assumptions from Subsection 6.5, our method is proved to be complete in generating a basis of the polynomial invariant ideal. For all examples presented in the thesis (Subsection 6.6) our approach returns a complete set of polynomial invariants. Moreover, we could not find any example of a P-solvable loop with conditional branches and assignments for which our approach fails to be complete. Thus we conjecture, that the imposed constraints in Subsection 6.5 cover a large class of imperative programs, and the completeness proof of our approach without the assumptions from Subsection 6.5 is a challenging task for further research.

The following subsections contain a detailed presentation of results presented in (Kovács and Jebelean, 2003b; Kovács *et al.*, 2003; Kovács and Jebelean, 2004a; Jebelean *et al.*, 2004) and their refinements.

5.3 P-solvable Loops with Assignments Only

As a "base"-case, in this section we present our polynomial invariant generation algorithm for *P-solvable loops with assignment statements only*.

Definition 5.9 An imperative loop (5.1) with assignment statements only is called *P-solvable* if the closed form system of its recursively changed variables $x_1, \ldots, x_m$ is

$$\begin{cases} x_1[n] = & p_{1,1}(n)\theta_1^n + \cdots + p_{1,s}(n)\theta_s^n \\ x_2[n] = & p_{2,1}(n)\theta_1^n + \cdots + p_{2,s}(n)\theta_s^n \\ \vdots & \\ x_m[n] = & p_{m,1}(n)\theta_1^n + \cdots + p_{m,s}(n)\theta_s^n \end{cases}, \tag{5.4}$$

where

(1) $x_i[n]$, $1 \leq i \leq m$, represents the value of x_i at iteration n;

(2) $p_{1,1}, \ldots, p_{1,s}, \ldots\ldots, p_{m,1}, \ldots, p_{m,s} \in \bar{\mathbb{K}}[n]$;

(3) $\theta_1, \ldots, \theta_s \in \bar{\mathbb{K}}$;

(4) there exist algebraic dependencies among $\theta_1^n, \ldots, \theta_s^n$.

The closed form expression from (5.4) is denoted by

$$\mathrm{CF}(S^n, E_S, X, X_0),$$

where

- S is the body of the P-solvable loop with assignments only (i.e. S is a sequence of assignments);
- $n \in \mathbb{N}$ is the loop counter;
- S^n denotes the n times repeated execution the loop body S, more precisely it denotes the sequence $\underbrace{S; \ldots; S}_{n \text{ times}}$;
- $E_S = \{\theta_1^n, \ldots, \theta_s^n\}$ is the set of algebraic sequences θ_i^n that occur in the closed form solutions of the loop variables;
- X and X_0 represent respectively the final and initial values of the loop variables after and before S^n.

Remark 5.10 W.l.o.g. we assumed that the number of exponential sequences in the closed form system (5.4) is the same, since

- we can always add terms $0 \cdot \theta_i^n$,
- by renaming, we can have the same number of exponential terms in each closed form.

Informally, an imperative loop is P-solvable if and only if the closed form solutions of the loop variables are linear combinations of exponential sequences $\theta_i^n \in \bar{\mathbb{K}}$ with polynomial coefficients $p(n) \in \bar{\mathbb{K}}[n]$, with the property that there exist algebraic dependencies among all θ_i^n. The coefficients of $p(n)$ are determined by the initial values X_0 of the loop variables before entering the loop.

The class of P-solvable loops includes the simple situation when the expressions in the assignment statements are affine mappings. Moreover, as presented in (4.12), any closed form solution of a loop variable defined by a C-finite recurrence is a linear combinations of terms $p_i(n)\theta_i^n$, where θ_i are the roots of the characteristic polynomial of the recurrence, and thus θ_i are polynomially related. Hence, a loop with only C-finite assignments (i.e. assignments that describe C-finite recurrence relations of the loop variables) is P-solvable.

Since finding closed form expressions of recurrences in the general case is undecidable (see e.g. (Schneider, 2004, 2005)), it is necessary to distinguish among the type of recurrence equations that occur in practical examples of imperative programs with loops.

In our work, we consider loops with the property that their assignment statements describe Gosper-summable or C-finite recurrences or mutually coupled recurrences that can be solved by the technique of generating functions.

5.4 Automated Polynomial Invariant Generation by Algebraic Techniques

Our method for automatically inferring polynomial relations as invariants is presented in Algorithm 5.11.

Algorithm 5.11 "receives" as input a P-solvable loop with assignment statements only, and starts first with extracting and solving the recurrence equations of the loop variables, determining this way the closed form solution system of the variables. These closed forms are linear combinations of exponential terms with polynomial coefficients in the loop counter (steps 1-3 of Algorithm 5.11).

Next, it computes the set A of generators for the ideal of algebraic dependencies among the exponential terms (step 4 of Algorithm 5.11).

Finally, from the ideal I generated by the polynomial system of closed forms and A, the ideal G of *all polynomial relations* among the loop variables is computed by elimination using Gröbner basis w.r.t. a suitable elimination order (steps 5-7 of Algorithm 5.11).

Algorithm 5.11 (P-solvable Loops with Assignments Only)
Input: Imperative P-solvable loop (5.1) with only assignment statements S, having its recursively changed variables $X = \{x_1, \ldots, x_m\}$ with initial values X_0
Output: The ideal $G \trianglelefteq \mathbb{K}[X]$ of polynomial invariants among the loop variables X
Assumption: The recurrence equations of the loop variables are of order greater or equal than 1

1 Extract the recurrence equations of the loop variables

I. Recurrence Solving.

2 Identify the type of recurrences and solve them using Algorithm 5.12
3 Using the P-solvable loop property, write the closed form system as

$$\mathrm{CF}(S^n, E_S, X, X_0): \begin{cases} x_1[n] = & p_{1,1}(n)\theta_1^n + \cdots + p_{1,s}(n)\theta_s^n \\ x_2[n] = & p_{2,1}(n)\theta_1^n + \cdots + p_{2,s}(n)\theta_s^n \\ \vdots & \\ x_m[n] = & p_{m,1}(n)\theta_1^n + \cdots + p_{m,s}(n)\theta_s^n \end{cases}, \text{ where } \begin{array}{l} \theta_j \in \bar{\mathbb{K}},\ p_{i,j} \in \bar{\mathbb{K}}[n], \\ p_{ij} \text{ are parameterized by } X_0, \\ j = 1, \ldots, s,\ \ i = 1, \ldots, m, \\ \\ E_S = \{\theta_1^n, \ldots, \theta_s^n\} \end{array}$$

4 Apply Algorithm 4.45 to identify the algebraic dependencies among $n, \theta_1^n, \ldots, \theta_s^n$.

$$\text{Denote } \langle A \rangle = I(n, \theta_1^n, \ldots, \theta_s^n)$$

5 Introduce the notations $y_0 = n, y_1 = \theta_1^n, \ldots, y_s = \theta_s^n$. Thus $\langle A \rangle \trianglelefteq \bar{\mathbb{K}}[y_0, \ldots, y_s]$ and

$$\begin{cases} x_1 = & q_1(y_0, y_1, \ldots, y_s) \\ x_2 = & q_2(y_0, y_1, \ldots y_s) \\ \vdots & \\ x_m = & q_m(y_0, y_1, \ldots, y_s) \end{cases}, \text{ where } \begin{array}{l} q_i \in \bar{\mathbb{K}}[y_0, y_1, \ldots, y_s],\ i = 1, \ldots, m, \\ q_i \text{ are parameterized by } X_0. \end{array}$$

II. Polynomial Invariant Generation.

6 Consider $I = \langle x_1 - q_1(y_0,\ldots,y_s),\ldots,x_m - q_m(y_0,\ldots,y_s)\rangle + \langle A\rangle$.

$$\text{Thus } I \subset \bar{\mathbb{K}}[y_0,y_1,\ldots,y_s,x_1,\ldots,x_m]$$

7 **return** $G = I \cap \mathbb{K}[x_1,\ldots,x_m]$.

Elimination of the variables $y_0,\ldots,y_s$ at step 7 is based on Algorithm 4.14. More specifically, Algorithm 4.8 is applied to compute the Gröbner basis of I w.r.t. an elimination order $\succ$ such that

$$y_0 \succ \cdots \succ y_s \succ x_1 \cdots \succ x_m.$$

Hence, Algorithm 5.11 computes the set of generators for the ideal G.

At step 2 of Algorithm 5.11 we use Algorithms 4.20 and 4.30 for handling Gosper-summable and C-finite recurrences, as well as Algorithm 4.38 for treating a coupled system of recurrences that can be solved by the technique of generating functions. The general solving algorithm at step 2 of Algorithm 5.11 is as follows.

Algorithm 5.12 (Solving Recurrences of P-solvable Loop's Variables)
Input: System of recurrences $\{rec_1,\ldots,rec_m\}$ for the values of the P-solvable loop variables $x_1[n],\ldots,x_m[n]$
Output: The closed form system for $x_1[n],\ldots,x_m[n]$
Assumption: rec_i, $i = 1,\ldots,m$, are of order greater or equal than 1

1 **repeat**
2 Identify and solve the recurrence equations of those loop variables that are independent of other recurrence equations by applying Algorithms 4.20 and 4.30
3 In case there is a mutually coupled recurrence system with constant coefficients, apply Algorithm 4.38 for obtaining the closed forms of the mutually dependent loop variables
4 Substitute the loop variables by their obtained closed forms in the remaining recurrence equations of the loop
5 **until** each closed form $x_i[n]$, $i = 1,\ldots,m$, is obtained
6 **return** the closed forms of $x_i[n]$, $i = 1,\ldots,m$.

Steps 2 and 3 of Algorithm 5.12 are crucial in our invariant generation algorithm. Namely, if the recurrence equations determined by the assignments of the loop cannot be solved using Algorithms 4.20, 4.30 or 4.38, the closed form solutions of the loop variables cannot be obtained, and thus Algorithm 5.11 fails in generating polynomial invariants. For instance, assignment statements describing recurrences as presented in Examples 4.22 or 4.25 show both the power and the limitations of the symbolic summation methods that our approach relies upon.

Thus, Algorithm 5.11 can be applied only to P-solvable loops. For these loops the existence of closed forms solutions of loop variables is guaranteed (see Definition 5.9).

Theorem 5.13 Algorithm 5.11 is correct. Its output G satisfies

(1) $G \trianglelefteq \mathbb{K}[x_1,\ldots,x_m]$;

(2) every polynomial relation from G is a polynomial invariant among the P-solvable loop variables $x_1,\ldots,x_m$ over $\mathbb{K}[X]$;

(3) any polynomial invariant among the P-solvable loop variables $x_1,\ldots,x_m$ over $\mathbb{K}[X]$ can be derived from (the generators of) G.

Proof.

1. Based on (5.4) and step 5 of Algorithm 5.11, for each P-solvable loop variable $x_i,\ i=1,\ldots,m$, we have

$$x_i = q_i(y_0,y_1,\ldots,y_s),\ \text{with } y_0 = n,\ y_j = \theta_j^n,\ j=1,\ldots,s.$$

Note that q_i is the closed form expression of variable x_i at iteration n of the loop.

Let us consider the polynomial map

$$\begin{aligned} &f : \bar{\mathbb{K}}[x_1,\ldots,x_m] \to \bar{\mathbb{K}}[y_0,y_1,\ldots,y_s]/\langle A\rangle,\\ &f(x_i) = q_i(y_0,y_1,\ldots,y_s) + \langle A\rangle, \qquad f(c) = c + \langle A\rangle \quad \text{for } c \in \bar{\mathbb{K}}. \end{aligned} \tag{5.5}$$

Using results of (Adams and Loustaunau, 1994) and Theorem 4.13, the ideal $I(x_1,\ldots,x_m)$ of polynomial relations among $x_1,\ldots,x_m$ is the kernel of the map f (as computed at steps 6-7 of Algorithm 5.11). Similarly to Algorithm 4.14, the kernel of f is computed by elimination using Gröbner basis w.r.t. a suitable elimination order, and we have

$$G = \ker f = I(x_1,\ldots,x_m) \trianglelefteq \bar{\mathbb{K}}[x_1,\ldots,x_m]. \tag{5.6}$$

Although the closed form solutions of $x_i,\ i=1,\ldots,m$, lie in the algebraic extension field $\bar{\mathbb{K}}$, note that the recurrences of x_i corresponding to the assignment statements of the loop are defined over $\mathbb{K}$. Based on (Kauers and Zimmermann (2006), Lemma 2 and Theorem 1), from (5.6) we thus derive

$$G = \ker f = I(x_1,\ldots,x_m) \trianglelefteq \mathbb{K}[x_1,\ldots,x_m]. \tag{5.7}$$

In other words, although the closed form solutions of $x_i,\ i=1,\ldots,m$, lie in the algebraic extension $\bar{\mathbb{K}}[y_0,y_1,\ldots,y_s]$, the (generators of the) ideal G of polynomial relations among the loop variables $x_1,\ldots,x_m$ is an ideal of $\mathbb{K}[x_1,\ldots,x_m]$, and we have

$$G \trianglelefteq \mathbb{K}[x_1,\ldots,x_m].$$

2. Let us take $p \in G$. Thus $p \in \ker f \trianglelefteq \mathbb{K}[x_1,\ldots,x_m]$, i.e. $f(p) = 0$, yielding that $f(p) \in \langle A\rangle$. We can write

$$p = \sum_e a_e x_1^{e_1}\cdots x_m^{e_m},$$

where $a_e \in \mathbb{K}$, $e = (e_1,\ldots,e_m) \in \mathbb{N}^m$, and only finitely many a_e's are non-zero.

Then $f(p) = \sum_e a_e q_1^{e_1}\cdots q_m^{e_m} = 0$. Thus, p becomes zero (i.e. it is in $\langle A\rangle$) whenever the closed forms q_i are substituted for the variables x_i in p, and hence p holds at any iteration n of the loop.

In particular it also holds at the entry of the loop, i.e. for $n = 0$. Therefore, p is polynomial invariant.

3. Let us consider an arbitrary polynomial invariant $u \in \mathbb{K}[X]$ among the P-solvable loop variables $x_1, \ldots, x_m$.
Since u is a polynomial invariant, it holds for any iteration n of the P-solvable loop. Hence, based on (5.4) and step 5 of Algorithm 5.11, u is in $\langle A \rangle = I(y_0, y_1, \ldots, y_s)$ whenever evaluated at $(x_1, \ldots, x_m) = (q_1, \ldots, q_m)$. Thus $u \in \ker f$, and we have

$$u \in G.$$

■

The restrictions at the various steps of Algorithm 5.11 are crucial. Namely, if the recurrences cannot be solved exactly, or their closed forms do not fulfill the P-solvable form, our algorithm fails in generating polynomial invariants among the loop variables.

Example 5.14 Given the imperative loop

$$\begin{array}{l} x := 10; y := 10; \\ \texttt{While}[y > 0, \\ \quad x := 2 * x; y := \frac{1}{2} * y - 1], \end{array}$$

its polynomial invariant by applying Algorithm 5.11 is obtained as follows.
$n \geq 0$, $x[0]$ and $y[0]$ denote the initial values of x and y before entering the loop.

Step 1:

$$\begin{cases} x[n+1] &=& 2 * x[n] \\ y[n+1] &=& \frac{1}{2} * y[n] - 1 \end{cases}.$$

Step 2,3:

$$\begin{cases} x[n] & \underset{C-finite}{=} & 2^n * x[0] \\ y[n] & \underset{C-finite}{=} & \frac{1}{2^n} * (y[0] - 2) + 2 \end{cases}.$$

The closed form system describes a P-solvable loop, and thus Algorithm 5.11 returns a basis for the ideal of polynomial invariants.

Step 4:
The generator of the ideal of algebraic dependencies of 2^n and 2^{-n} is

$$A = \{2^n * 2^{-n} - 1\}.$$

Step 5:
$a = 2^n,\ b = 2^{-n},\ c = n.$

$$\begin{cases} x &=& a * x[0] \\ y &=& b * (y[0] - 2) + 2 \\ 0 &=& a * b - 1 \end{cases}.$$

Step 6:

$$I = \langle x - a * x[0],\ y - b * (y[0] - 2) + 2,\ a * b - 1 \rangle.$$

Step 7: The computed Gröbner basis of I w.r.t $a \succ b \succ x \succ y$ is

$$\begin{aligned} J = \quad & \{2*x+x*y+2*x[0]-x[0]*y[0], \\ & -2-2*b-y+b*y[0], \\ & b*x-x[0], \\ & -x+a*x[0], \\ & 2+2*a+a*y-y[0], \\ & -1+a*b\}. \end{aligned}$$

By $J \cap \mathbb{K}[x,y]$ the obtained polynomial invariant is

$$2*x+x*y+2*x[0]-x[0]*y[0].$$

Further, the symbolically treated initial values $x[0]$ and $y[0]$ might be substituted by their values given at the loop entry. We thus obtain

$$2*x+x*y-120.$$

Example 5.15 For Example 5.1, by applying Algorithm 5.11, the polynomial invariants are obtained as follows.

$n \geq 0$, $quo[0]$ and $rem[0]$ denote the initial values of quo and rem before entering the loop.

Step 2, 3:

$$\begin{cases} quo[n] & \underset{Gosper}{=} & quo[0]+n \\ rem[n] & \underset{Gosper}{=} & rem[0]-n*y \end{cases}.$$

Considering the exponentials 1^n, the closed form system describes a P-solvable loop and thus Algorithm 5.11 returns a basis for the ideal of polynomial invariants.

Step 4, 5, 6:

$$\begin{aligned} & a = n. \\ & I = \langle \\ & \quad quo - quo[0] - a, \\ & \quad rem - rem[0] - a*y, \\ & \rangle. \end{aligned}$$

Step 7:

By the computation of Gröbner basis with the elimination order $n \succ rem \succ quo$, the obtained polynomial invariant is

$$rem + quo*y - y*quo[0] - rem[0].$$

Further, the symbolically treated initial values $quo[0]$ and $rem[0]$ might be substituted by their values given at the loop entry. We thus obtain

$$rem + quo*y - x,$$

which corresponds to the polynomial invariant of the loop invariant stated in Example 2.4.

5.5 P-solvable Loop Properties

Theorem 5.16

Given an imperative loop with assignment statements only and loop counter $n \geq 0$.
If the closed form system of the recursively changed loop variables $\{x_1, \ldots, x_m\}$ is as (5.4) with the property that there are no algebraic dependencies among the exponential sequences $\theta_i^n,\ i = 1, \ldots, s$, then the loop has no polynomial invariant.

Proof. We prove by contradiction.
Consider a loop with closed form system (5.4) of its variables, with the property that there are no algebraic dependencies among $\theta_1^n, \ldots, \theta_s^n$. We introduce the notations

$$y_0 = n,\ y_1 = \theta_1^n,\ y_s = \theta_s^n,$$

and thus, by Theorem 4.43, we have

$$I(y_0, y_1, \ldots, y_s) = \emptyset \trianglelefteq \bar{\mathbb{K}}[y_0, y_1, \ldots, y_s].$$

Using the above notations, similarly to step 5 of Algorithm 5.11, from the closed form system (5.4) of the loop variables we derive

$$\begin{cases} x_1 = & q_1(y_0, y_1, \ldots, y_s) \\ x_2 = & q_2(y_0, y_1, \ldots y_s) \\ \vdots & \\ x_m = & q_m(y_0, y_1, \ldots, y_s) \end{cases}, \tag{5.8}$$

where $q_i \in \bar{\mathbb{K}}[y_0, y_1, \ldots, y_s],\ i = 1, \ldots, m$.
Let's assume that the loop has non-trivial polynomial invariants. Then there are polynomial relations among $x_1, \ldots, x_m$. We denote the ideal of polynomial relations among $x_1, \ldots, x_m$ by

$$I := I(x_1, \ldots, x_m) \trianglelefteq \mathbb{K}[x_1, \ldots, x_m], \text{ and thus } I \neq \emptyset.$$

Hence, we have

$$\exists_g\ g \in I \trianglelefteq \mathbb{K}[x_1, \ldots, x_m].$$

Using (5.8), g is thus in $I(y_0, y_1, \ldots, y_s)$ whenever evaluated at $(x_1, \ldots, x_m) = (q_1, \ldots, q_m)$, contradicting the assumption that $I(y_0, y_1, \ldots, y_s) = \emptyset$.
Therefore, if an imperative loop admits no algebraic dependencies among the exponential sequences present in the closed form (5.4) of its variables, then the loop has no polynomial invariant.

■

In the process of automatically inferring polynomial invariants for P-solvable loops, efficient techniques from symbolic summations are involved. Hence, Algorithm 5.11 can be applied on a rich class of practical imperative loops with assignments. From these assignments, the values of

variables are expressed in terms of their previously computed values, and the loop is thus modeled by means of recurrence equations.

Moreover, using recurrence equations of order greater or equal to 2, we are able to handle imperative loops where auxiliary variables are used in order to refer to values of variables computed at finitely many loop iterations before (the number of previous loop iterations is given by the order of the recurrence). Such a loop behavior is presented below.

Proposition 5.17 Given the P-solvable loop

$$\begin{array}{l}\texttt{While[...,}\\ \quad t := r;\\ \quad r := b*r + a*q;\\ \quad q := t;\\ \quad x := a*x\\ \texttt{],}\end{array}$$

where $t,\ r,\ q,\ x$ are loop variables and $a,\ b \in \mathbb{K}$ are absolute constants, i.e. they do not depend on loop variables.

Then there is always a polynomial invariant $p \in \mathbb{K}[r,\ q,\ x]$ among r, q and x.

Proof. Denoting by $n \geq 0$ the loop counter, the given imperative loop can be expressed in terms of recurrence equations as follows.

$$\left\{\begin{array}{lcl} r[n+2] &=& b*r[n+1] + a*r[n]\\ q[n+2] &=& r[n+1]\\ x[n+2] &=& a*x[n+1]\end{array}\right. .$$

Note that the recurrence of r is of order 2. Therefore, in the closed form computation of r two initial values of r are needed (that are the initial values of r and q before entering the loop). Thus, the one-to-one correspondence between the loop counter and the recurrence index does not hold anymore, but, denoting by j the recurrence counter, we have $j = n+1$, where $j \geq 1$ and $n \geq 0$.

Hence, we can apply Algorithm 4.30 to obtain the system of closed form solutions (in j, and thus also in n) of the loop variables. This can be one of the 2 possibilities, depending on the values of $a,\ b \in \mathbb{K}$.

Case 1: $y_1 = y_2$

$$\left\{\begin{array}{ll} r[j] = & \alpha*y_1^j + \beta*j*y_1^j\\ q[j] = & r[j-1]\\ x[j] = & a^j x[0]\end{array}\right.$$

Case 2: $y_1 \neq y_2$

$$\left\{\begin{array}{ll} r[j] = & \alpha*y_1^j + \beta*y_2^j\\ q[j] = & r[j-1]\\ x[j] = & a^j x[0]\end{array}\right. ,$$

with

$$\left\{\begin{array}{lcl} \alpha &=& q[0]\\ \alpha*y_1 + \beta*y_1 &=& r[0]\end{array}\right.$$

$$\left\{\begin{array}{lcl} \alpha + \beta &=& q[0]\\ \alpha*y_1 + \beta*y_2 &=& r[0]\end{array}\right.$$

where $r[0],\ q[0],\ x[0]$ denote the initial values of $r,\ q,\ x$ before entering the loop.

y_1 and y_2 are the algebraic roots (rational or non-rational algebraic numbers) of the characteristic polynomial corresponding to the recurrence equation of r, and thus

$$y_1 * y_2 = a.$$

Hence, by Algorithm 4.45 there are algebraic dependencies among y_1^j, y_2^j, a^j, namely, we have

$$(y_1^j)^2 * (y_2^j)^2 - (a^j)^2 = 0.$$

Since the algebraic dependencies and closed form solutions hold for all $j \geq 1$, they also hold for all loop iterations $n \geq 0$ and thus the given loop is P-solvable.

Applying Algorithm 5.11, the obtained polynomial relation among the loop variables r, q, x is

$$\{-a^2\, x^2\, q[0]^4 - 2\, a\, b\, x^2\, q[0]^3\, r[0] + 2\, a\, x^2\, q[0]^2\, r[0]^2 - b^2\, x^2\, q[0]^2\, r[0]^2 + 2\, b\, x^2\, q[0]\, r[0]^3 - x^2\, r[0]^4 + a^2\, q^4\, x[0]^2 + 2\, a\, b\, q^3\, r\, x[0]^2 - 2\, a\, q^2\, r^2\, x[0]^2 + b^2\, q^2\, r^2\, x[0]^2 - 2\, b\, q\, r^3\, x[0]^2 + r^4\, x[0]^2\}.$$

■

5.6 A Decidable Class: Affine Loops

Based on (4.12) and Definition 4.33, any closed form solution of a loop variable defined by an affine relation among the loop variables is a linear combination of polynomials and algebraically related exponentials in the summation variable (and thus, in the loop counter). For such cases the closed form requirement of the P-solvable loop from Definition 5.9 is hence fulfilled.

In this subsection we study a special class of P-solvable loops, called the *affine loops* (see Definition 5.19). For such loops we show that our method is complete in inferring the generators for the ideal of polynomial invariants. Moreover, by results of Subsection 5.3, our approach offers an algorithmic computation of the basis for the ideal of all polynomial invariants among the variables of the affine loop.

Definition 5.18 An *affine assignment* of a program variable $x_i \in X = \{x_1, \ldots, x_m\}$ is

$$x_i = \sum_{k=1}^{m} a_k * x_k + b_i, \tag{5.9}$$

where $a_k, b_i \in \mathbb{K},\ i = 1, \ldots, m$.

Based on Definition 4.33, affine assignments of an imperative loop can be expressed by affine relations. Thus, several interesting properties of imperative loops with affine assignments only can be derived by symbolic summation algorithms.

Definition 5.19 An imperative loop with only affine assignments is called an *affine loop*.

Thus, considering $n \geq 0$ as the loop counter, the variables $X = \{x_1, \ldots, x_m\}$ of an affine loop satisfy the matrix equation

$$X[n+1] = A * X[n] + B, \tag{5.10}$$

with the matrix $A \in \mathbb{K}^{m \times m}$ and the (column) vector $B \in \mathbb{K}^m$.

Expanding (5.10), the affine assignments of the loop variables are defined by the affine relations

$$x_i[n+1] = \sum_{k=1}^{m} a_{ik} * x_k[n] + b_i,\ a_{ik}, b_i \in \mathbb{K},\ i = 1, \ldots m. \tag{5.11}$$

Theorem 5.20 An affine loop is a P-solvable loop.

Proof. Consider the matrix equation of the affine loop's variables as in (5.10), namely:

$$X[n+1] = A * X[n] + B,$$

with the matrix $A \in \mathbb{K}^{m \times m}$ and the (column) vector $B \in \mathbb{K}^m$.

Thus, we have the inhomogeneous linear recurrences

$$x_i[n+1] = \sum_{k=1}^{m} a_{ik} * x_k[n] + b_i,\ a_{ik}, b_i \in \mathbb{K},\ i = 1, \ldots m. \tag{5.12}$$

We denote by

$$y_i(n) = b_i,$$

and consider the shift operator S_i, defined by (4.9). We thus have

$$(S_i - 1) \cdot y_i = 0,$$

that, by (5.12), is

$$(S_i - 1) \cdot (x_i[n+1] - \sum_{k=1}^{m} a_{ik} * x_k[n]) = 0.$$

Hence

$$(x_i[n+2] - \sum_{k=1}^{m} a_{ik} * x_k[n+1]) - (x_i[n+1] - \sum_{k=1}^{m} a_{ik} * x_k[n]) = 0, \tag{5.13}$$

and thus, in a similar way as presented in Proposition 4.32, the inhomogeneous parts of the C-finite sequences $x_i[n]$ are canceled out. Based on (5.12) and Definition 4.28, (5.13) describes a C-finite system of coupled C-finite sequences. For C-finite coupled systems there are known "uncoupling" algorithms which transform the system into an equivalent one consisting only of independent C-finite recurrences (Zürcher, 1994; Gerhold, 2002).

Hence, the coupled C-finite system (5.13) can be reduced to a system of independent C-finite recurrences on which we can then apply Algorithms 4.30 and 4.35 to obtain closed form solutions of $x_i[n]$.

We thus have the system of closed form solutions given below.

$$\left\{ \begin{array}{lcl} x_1[n] & = & p_{1,1}(n)\theta_1^n + \cdots + p_{1,s}(n)\theta_s^n \\ x_2[n] & = & p_{2,1}(n)\theta_1^n + \cdots + p_{2,s}(n)\theta_s^n \\ & \vdots & \\ x_m[n] & = & p_{m,1}(n)\theta_1^n + \cdots + p_{m,s}(n)\theta_s^n \end{array} \right., \tag{5.14}$$

where $\theta_j,\ j = 1, \ldots, s$, are the roots of the characteristic polynomials of C-finite recurrences corresponding to the loop variables. θ_j^n are thus algebraically related algebraic exponential sequences whose algebraic dependencies are computed by Algorithm 4.45.

By (5.4), from (5.14) we finally have that the affine loop is P-solvable.

■

Remark 5.21 Results of (Zürcher, 1994; Gerhold, 2002) can deal with more general cases, namely, to uncouple a coupled linear recurrence system with polynomial coefficients in the summation index. Thus, a C-finite coupled system is a special case of the power of their algorithms.

Based on Theorem 5.20 and 5.16, we finally state the theorem below.

Theorem 5.22 The ideal of all polynomial invariants for an affine loop is algorithmically computable by Algorithm 5.11.

Proof. It follows from Theorem 5.20 and the correctness of Algorithm 5.11.

■

5.7 Related Work on Reasoning about Affine Loops

Finding valid affine relations, i.e. polynomial relations of degree 1, has many applications (Karr, 1976), (Müller-Olm and Seidl, 2004a): many classical data flow analysis problems can be treated as problems about affine relations. For instance, in the case of *constant propagation*, derivation of relations like $x = c$ is needed, that is a special case of affine relationships among variables, where x is a program variable and c is a constant from the domain corresponding to the type of x. More generally, by automatically inferring valid affine relationships, one could recognize whether *linear expressions* in program variables are constant, e.g. $2*x+3*y = 1$, where x, y are program variables, even though their subexpressions might not be. This would thus also yield detection of so-called *definite equalities among variables*, e.g. $x = y$. Another application is the recognition of common affine subexpressions which are formally not identical, but are the same (same value at run-time) because of the expressed relationship among the variables. Moreover, valid affine relations among loop variables can be used as program assertions in program verification, as they express non-trivial relations among variables, and thus simplify the verification task.

In 1976, M. Karr proposed a general technique for finding affine relationships among program variables (Karr, 1976). His forward-propagating algorithm determines for every program point the set of *all* valid affine relations by manipulations of affine subspaces in which the relations lie. Thus, a finite representation of these affine spaces is needed. Karr (1976) chooses to represent the affine spaces as kernels of affine transformations, i.e. as sets of solutions of systems of linear equations. However, Karr's work used quite complicated operations (transformations on invertible/non-invertible assignments, affine union of spaces) and had a limitation on arithmetical operations among the program variables. For these reasons, extension of his work has recently become a challenging research topic (Müller-Olm and Seidl, 2002, 2004b,c; Gulwani and Necula, 2003; Rodriguez-Carbonell and Kapur, 2004, 2007a).

Based on results of (M. Müller Olm and O. Rüthing, 2001), in (Müller-Olm and Seidl, 2002) a framework for automatically checking polynomial constants of the form $x - c = 0$ at a program point by *backward propagation* is described. The approach has been extended in (Müller-Olm and Seidl, 2004b,c; Müller-Olm *et al.*, 2006) for checking and, most importantly, deriving valid polynomial relations of bounded degree among the program variables. This is performed by computing algorithmically the polynomial ideal that represents the weakest precondition at the target point of a *generic polynomial relation* of an a priori fixed degree d. The coefficients of this

polynomial are replaced by variables. By Hilbert's basis theorem and Buchberger's algorithm (Buchberger, 2006), the ideal representing the weakest precondition can be computed effectively. Since the polynomial relation in question is valid at the target point if and only if its weakest precondition holds for all states, the set of values for the coefficients of the polynomial is characterized by a linear equation system (constraint system). The solution space of this system characterizes the coefficients of all polynomial relations up to the chosen degree d that are valid at the target point.

Based on this general setting, the derivation of *all affine* relations (polynomial relations of degree 1) among the variables of polynomial programs, and hence in the case of affine programs as well, can be performed effectively as discussed in (Müller-Olm and Seidl, 2004a). However, by their approach, finding *all polynomial* relations of affine (or polynomial) programs is possible only up to an a priori chosen degree. In the case when the program has valid polynomial relations of different degrees, the approach has to be applied separately for each degree.

This is not the case of our algorithm. Our restriction is not on the degree of sought polynomial relations, but on the type of assignments (recurrence equations) present in the loop body. The shape of assignments restricts our approach to the class of P-solvable loops, and thus we cannot handle loops with arbitrary polynomial assignments. However, for P-solvable loops with assignments only our method returns the generator set of all polynomial relations (of different degrees) among the program variables by the application of our method only once. Further, as shown in Subsection 5.6, for the case of loops with affine assignments only, our method is complete in generating polynomial relations from which any other polynomial invariant can be derived. Moreover, as presented in Section 6, for P-solvable loops with conditional branches and assignments, our approach returns a set of polynomial invariants, that, under additional assumptions, represents a basis for the polynomial invariant ideal of the loop.

By fixing also the degree of sought polynomial invariants, in (Rodriguez-Carbonell and Kapur, 2007a) a method based on the framework of abstract interpretation is presented for generation of polynomial invariants for loops with polynomial assignments. The method also handles nested loops, as well as test conditions in conditional statements and loops if these tests are expressed using polynomial equalities and disequalities. Moreover, if the tests are ignored and all assignments in the loop are linear, the approach finds all polynomial invariants of the fixed degree.

Contrarily to (Rodriguez-Carbonell and Kapur, 2007a), by considering loops with only linear assignments (i.e. affine assignment) and conditional branches with ignored tests, under additional structural assumptions over ideals of polynomial relations for the case of loop with conditional branches, our method generates a basis of the polynomial invariant ideal without fixing a priori a polynomial degree. However, in comparison to (Rodriguez-Carbonell and Kapur, 2007a), we do not yet handle nested loops and the test conditions are always ignored, since, by results of (Müller-Olm and Seidl, 2004a), the ideal of all polynomial invariants is not computable if polynomial equality tests are considered.

(Rodriguez-Carbonell and Kapur, 2007b) introduces *simple loops*, for which they present a method for generating *all* polynomial invariants, instead of fixed degree polynomials used in (Rodriguez-Carbonell and Kapur, 2007a). A loop is said to be *simple* if it contains only conditionals and assignments, the tests in the loop and conditional statements are ignored, and assignments are

solvable mappings with positive rational eigenvalues. For such loops a fixpoint procedure for invariant generation is presented by iteratively computing the ideal of polynomial relations among the loop variables using the theory of Gröbner basis, until a fixpoint is reached. This fixpoint is the ideal of polynomial invariants.

The concept of *solvable mapping* refers to the solvability of the recurrence equation defined by the assignment in question. Solvable mappings are recursively built using blocks of variables, starting with a block of variables whose assignments are expressed as affine transformations. Thus, affine mappings are a special class of solvable mappings. The restriction of assignment mappings being solvable with positive rational eigenvalues ensures that the program variables can be polynomially expressed in terms of the loop counter and some auxiliary *rational* variables. Hence, the concept of solvable mapping is similar to the definition of P-solvable loop.

However, contrarily to (Rodriguez-Carbonell and Kapur, 2007b), in our approach we compute closed form solutions of program variables for a wider class of recurrence equations (assignment statements) based on several techniques from symbolic summation, e.g. Gosper algorithm that allows handling of hypergeometric sums. The restriction on the closed form solution for P-solvable loops brings our approach also to the case of having closed forms as polynomials in the loop counters and additional new variables, but, unlike (Rodriguez-Carbonell and Kapur, 2007b), the new variables can be arbitrary *algebraic* numbers, and not just rationals. Thus, our method handles affine loops in a complete manner without any restriction on the assignments (see Theorem 5.20), whereas the algorithmic computation of polynomial invariants based on (Rodriguez-Carbonell and Kapur, 2007b) requires positive rational eigenvalues in generating a complete set of polynomial invariants. Hence, as also shown by e.g. Examples 5.24, 5.29, 5.30, the class of *simple loops* with assignments only is a subclass of *P-solvable* loops.

Further, for the case of P-solvable loops with conditional branches and assignments, by the considered techniques from algorithmic combinatorics, our approach can handle a richer class of assignments statements in the loop body, such that at the end of the invariant generation process, polynomial invariants of the P-solvable loop are obtained. However, contrarily to (Rodriguez-Carbonell and Kapur, 2007b) where completeness is always guaranteed, the completeness of our method for loops with conditional branches and assignments is proved only under additional assumptions over ideals of polynomial invariants. It is worth to be mentioned though that these additional constraints cover a wide class of loops, and we could not find any example for which the completeness of our approach is violated.

5.8 Examples

All results of the examples presented here are outputs of our software package `Aligator` for **a**utomated **l**oop **i**nvariant **g**eneration by **a**lgebraic **t**echniques **o**ver the **r**ationals, implemented in *Mathematica*.

The examples below illustrate the steps of our algorithm for generating a basis of the polynomial invariant ideal of P-solvable loops with assignment statements only. They also illustrate the power and limitations of our method.

Detailed Examples

Example 5.23 Given the imperative loop

$$
\begin{array}{l}
r := 2; q := 1; x := 2; \\
\texttt{While}[\ldots, \\
\quad t := r; \\
\quad r := 2 * r - 8 * q; \\
\quad q := t; \\
\quad x := 8 * x \\
],
\end{array}
$$

the loop body is described by the recurrence equations

$$
\left\{
\begin{array}{lll}
r[n+2] = & 2 * r[n+1] - 8 * r[n] \\
q[n+2] = & r[n+1] \\
x[n+2] = & 8 * x[n+1]
\end{array}
\right. ,
$$

where $n \geq 0$ denotes the loop iteration counter.

Similarly to Proposition 5.17, note that the recurrence of r is of order 2, meaning that for solving it two initial values of r are needed (that are the initial values of r and q before entering the loop). Thus, the one-to-one correspondence between the loop counter and the recurrence index does not hold anymore. Denoting by j the recurrence counter, we have $j = n+1$, where $j \geq 1$ and $n \geq 0$.

By Definition 4.24, the recurrences of r and x are C-finite, thus by Algorithm 4.30 we obtain the closed form system in j (where, conform step 5 of Algorithm 5.11, r, q, x stand for $r[j]$, $q[j]$, $x[j]$), as presented below.

$$
\left\{
\begin{array}{lll}
r = & a * y_1 + b * y_2 \\
q = & \frac{1}{1-\mathrm{i}\sqrt{7}} * a * y_1 + \frac{1}{1+\mathrm{i}\sqrt{7}} * b * y_2 \\
x = & \frac{1}{8} * y_3 * x[0]
\end{array}
\right. ,
$$

where

$$
\left\{
\begin{array}{l}
y_1 = (1-\mathrm{i}\sqrt{7})^j, \quad y_2 = (1+\mathrm{i}\sqrt{7})^j, \quad y_3 = 8^j \\
q[0] = a+b, \quad r[0] = a * (1-\mathrm{i} * \sqrt{7}) + b * (1+\mathrm{i} * \sqrt{7})
\end{array}
\right. ,
$$

where $r[0], q[0], x[0]$ are the initial values of the loop variables r, q, x before entering the loop.

The algebraic dependencies among y_1, y_2, y_3 can be computed by Algorithm 4.45 and we have

$$y_1 * y_2 - y_3 = 0.$$

The algebraic dependencies and closed forms hold for all $j \geq 1$, and thus for all $n \geq 0$ loop iterations, yielding that the given imperative loop is P-solvable. Hence, we determine the polynomial

invariants of the loop by first considering the ideal

$$\begin{aligned} I = \langle \; & \\ & r - a * y_1 - b * y_2, \\ & q - \tfrac{1}{1-\mathrm{i}*\sqrt{7}} * a * y_1 - \tfrac{1}{1+\mathrm{i}*\sqrt{7}} * b * y_2, \\ & x - \tfrac{1}{8} * y_3 * x[0], \\ & q[0] - a - b, \\ & r[0] - a * (1 - \mathrm{i} * \sqrt{7}) * (1 + \mathrm{i} * \sqrt{7}), \\ & y_1 * y_2 - y_3 \\ \rangle. & \end{aligned}$$

Finally performing the Gröbner basis computation for the elimination ideal, using $n > y_1 > y_2 > y_3 > a > b > r > q > x$, the obtained polynomial invariant (the output of `Aligator`) is

$$\{-8\,x\,q[0]^2 + 2\,x\,q[0]\,r[0] - x\,r[0]^2 + 8\,q^2\,x[0] - 2\,q\,r\,x[0] + r^2 x[0]\},$$

yielding

$$\{8q^2 - 2qr + r^2 - 4x\},$$

by initial values substitutions.

Example 5.24 Given the imperative loop

```
b := 5/16; a := 3/4; r := 2;
While[...,
    s := t;
    t := r;
    r := 5 * r - 7 * a + b;
    a := t;
    b := s
],
```

the loop body is described by the recurrence equations

$$\begin{cases} r[n+3] = & 5 * r[n+2] - 7 * r[n+1] + r[n] \\ a[n+3] = & r[n+2] \\ b[n+3] = & r[n+1] \end{cases},$$

where $n \geq 0$ denotes the loop iteration counter.

Similarly to Proposition 5.17, note that the recurrence of r is of order 3, meaning that for solving it three initial values of r are needed (that are the initial values of r, a and b before entering the loop). Thus, the one-to-one correspondence between the loop counter and the recurrence index

does not hold anymore. Denoting by j the recurrence counter, we have $j = n+2$, where $j \geq 2$ and $n \geq 0$.

By Definition 4.24, the recurrence of r is C-finite, thus by Algorithm 4.30 we determine the closed form system of $r,\ a,\ b$ in $j \geq 2$. Conform step 5 of Algorithm 5.11, $r,\ a,\ b$ stand for $r[j],\ a[j],\ b[j]$, and we denote by $r[0], a[0], b[0]$ the initial values of r, a, b before entering the loop. Thus the closed form system of the loop variables is as follows.

$$\begin{cases} r = & u*y_1 + v*y_2 + t*y_3 \\ a = & \frac{1}{z_1}*u*y_1 + \frac{1}{z_2}*v*y_2 + \frac{1}{z_3}*t*y_3 \\ b = & \frac{1}{z_1^2}*u*y_1 + \frac{1}{z_2^2}*v*y_2 + \frac{1}{z_3^2}*t*y_3 \end{cases},$$

where

$$\begin{cases} y_1 = \left(\frac{1}{3}\left(5 - \frac{4}{(19-3*\sqrt{33})^{\frac{1}{3}}} - (19-3\sqrt{33})^{\frac{1}{3}}\right)\right)^j \\ y_2 = \left(\frac{5}{3} + \frac{2*(1+\mathrm{i}*\sqrt{3})}{3*(19-3*\sqrt{33})^{\frac{1}{3}}} + \frac{1}{6}*(1-\mathrm{i}*\sqrt{3})*(19-3*\sqrt{33})^{\frac{1}{3}}\right)^j \\ y_3 = \left(\frac{5}{3} + \frac{2*(1-\mathrm{i}*\sqrt{3})}{3*(19-3*\sqrt{33})^{\frac{1}{3}}} + \frac{1}{6}*(1+\mathrm{i}*\sqrt{3})*(19-3*\sqrt{33})^{\frac{1}{3}}\right)^j \\ z_1 = \frac{1}{3}\left(5 - \frac{4}{(19-3*\sqrt{33})^{\frac{1}{3}}} - (19-3\sqrt{33})^{\frac{1}{3}}\right) \\ z_2 = \frac{5}{3} + \frac{2*(1+\mathrm{i}*\sqrt{3})}{3*(19-3*\sqrt{33})^{\frac{1}{3}}} + \frac{1}{6}*(1-\mathrm{i}*\sqrt{3})*(19-3*\sqrt{33})^{\frac{1}{3}} \\ z_3 = \frac{5}{3} + \frac{2*(1-\mathrm{i}*\sqrt{3})}{3*(19-3*\sqrt{33})^{\frac{1}{3}}} + \frac{1}{6}*(1+\mathrm{i}*\sqrt{3})*(19-3*\sqrt{33})^{\frac{1}{3}} \\ b[0] = u+v+t \\ a[0] = u*z_1 + v*z_2 + t*z_3 \\ r[0] = u*z_1^2 + v*z_2^2 + t*z_3^2 \end{cases}.$$

The algebraic dependencies among y_1, y_2, y_3 can be computed by Algorithm 4.45 and we have

$$1 - y_1 * y_2 * y_3 = 0.$$

The computed closed forms and algebraic dependencies hold for all $j \geq 2$, and thus for all $n \geq 0$ loop iterations, yielding that the loop is P-solvable. Thus, we determine the polynomial invariants

of the loop by first considering the ideal

$$
\begin{aligned}
I = \langle\ & \\
& r-(u*y_1+v*y_2+t*y_3),\\
& a-(\tfrac{1}{z_1}*u*y_1+\tfrac{1}{z_2}*v*y_2+\tfrac{1}{z_3}*t*y_3),\\
& b-(\tfrac{1}{z_1^2}*u*y_1+\tfrac{1}{z_2^2}*v*y_2+\tfrac{1}{z_3^2}*t*y_3),\\
& b[0]-u-v-t,\\
& a[0]-u*z_1-v*z_2-t*z_3,\\
& r[0]-u*z_1^2-v*z_2^2-t*z_3^2,\\
& z_1-\Big(\tfrac{1}{3}\big(5-\tfrac{4}{(19-3*\sqrt{33})^{\frac{1}{3}}}-(19-3\sqrt{33})^{\frac{1}{3}}\big)\Big),\\
& z_2-\Big(\tfrac{5}{3}+\tfrac{2*(1+\mathrm{i}*\sqrt{3})}{3*(19-3*\sqrt{33})^{\frac{1}{3}}}+\tfrac{1}{6}*(1-\mathrm{i}*\sqrt{3})*(19-3*\sqrt{33})^{\frac{1}{3}}\Big),\\
& z_3-\Big(\tfrac{5}{3}+\tfrac{2*(1-\mathrm{i}*\sqrt{3})}{3*(19-3*\sqrt{33})^{\frac{1}{3}}}+\tfrac{1}{6}*(1+\mathrm{i}*\sqrt{3})*(19-3*\sqrt{33})^{\frac{1}{3}}\Big),\\
& 1-y_1*y_2*y_3\\
\rangle.&
\end{aligned}
$$

Finally performing the Gröbner basis computation for the elimination ideal, using $n > y_1 > y_2 > y_3 > u > v > t > z_1 > z_2 > z_3 > r > a > b$, the obtained polynomial invariant (the output of `Aligator`) is

$$
\begin{aligned}
&\{-34\,a^3+54\,a^2\,b-14\,a\,b^2+b^3+32\,a^2\,r-38\,a\,b\,r+5\,b^2\,r-10\,a\,r^2+7\,b\,r^2+r^3+\\
&34\,a[0]^3-54\,a[0]^2\,b[0]+14\,a[0]\,b[0]^2-b[0]^3-32\,a[0]^2\,r[0]+38\,a[0]\,b[0]\,r[0]-\\
&5\,b[0]^2\,r[0]+10\,a[0]\,r[0]^2-7\,b[0]\,r[0]^2-r[0]^3\}.
\end{aligned}
$$

Further, by initial values substitutions, we obtain

$$
\begin{aligned}
&\{-277-139264\,a^3+221184\,a^2b-57344\,a*b^2+4096\,b^3+131072\,a^2r-155648\,abr+\\
&20480\,b^2r-40960\,ar^2+28672\,br^2+4096\,r^3\}.
\end{aligned}
$$

Further Examples - only with `Aligator`'s final results

Example 5.25 Given the imperative loop

$$
\begin{array}{l}
r := 1;\ a := 1;\ b := 0;\\
\texttt{While}[\ldots,\\
\quad s := t;\\
\quad t := r;\\
\quad r := r + a + b;\\
\quad a := t;\\
\quad b := s\\
].
\end{array}
$$

By applying Algorithm 5.11 and using `Aligator`, we obtain the polynomial invariant

$$\{2\,a^3 + 2\,a^2\,b + 2\,a\,b^2 + b^3 - 2\,a\,b\,r + b^2\,r - 2\,a\,r^2 - b\,r^2 + r^3 - 2\,a[0]^3 - 2\,a[0]^2\,b[0] - 2\,a[0]\,b[0]^2 - b[0]^3 + 2\,a[0]\,b[0]\,r[0] - b[0]^2\,r[0] + 2\,a[0]\,r[0]^2 + b[0]\,r[0]^2 - r[0]^3\},$$

that is

$$\{-1 + 2a^3 + 2a^2b + 2ab^2 + b^3 - 2abr + b^2r - 2ar^2 - br^2 + r^3\},$$

by initial values substitutions.

Example 5.26 Given the imperative loop

$$
\begin{array}{l}
x := 1;\ y := 0;\\
\texttt{While}[\ldots,\\
\quad x := 2 * x;\\
\quad y := 1/2 * y + 1].
\end{array}
$$

By applying Algorithm 5.11, using `Aligator`, we obtain the polynomial invariant

$$\{-2\,x + x\,y + 2\,x[0] - x[0]\,y[0]\},$$

yielding

$$\{2 - 2x + xy\},$$

by initial values substitutions.

Example 5.27 Given the imperative loop

$$
\begin{array}{l}
a := 0;\ b := 0;\ c := 1;\ s := 0;\\
\texttt{While}[\ldots,\\
\quad a := a + 1;\\
\quad b := b + c;\\
\quad c := c + 2;\\
\quad s := s + 2 * a + 1].
\end{array}
$$

By applying Algorithm 5.11 and using `Aligator`, we obtain the polynomial invariants

$$\{-4\,c-c^2+4\,s-4\,c\,a[0]+4\,c[0]+2\,c\,c[0]+4\,a[0]\,c[0]-c[0]^2-4\,s[0],$$
$$2\,b+3\,c-2\,s+2\,c\,a[0]-2\,b[0]-3\,c[0]-c\,c[0]-2\,a[0]\,c[0]+c[0]^2+2\,s[0],$$
$$2a-c-2\,a[0]+c[0]\},$$

yielding

$$\{\ -3+2c+c^2-4s,\ -1+b+c-s,\ 1+2a-c\ \},$$

by initial values substitutions.

Example 5.28 Given the imperative loop

$$n := 0;\ x := 0;\ y := 1;\ z := 6;$$
$$\texttt{While}[(n \leq N),$$
$$\quad x := x+y;$$
$$\quad y := y+z;$$
$$\quad z := z+6;$$
$$\quad n := n+1].$$

By applying Algorithm 5.11 and using `Aligator`, we obtain the polynomial invariants

$$\{-6\,n+z+6\,n[0]-z[0],$$
$$3\,n-3\,n^2+y-3\,n[0]+6\,n\,n[0]-3\,n[0]^2-y[0]-n\,z[0]+n[0]\,z[0],$$
$$-4\,n+6\,n^2-2\,n^3+2\,x+4\,n[0]-12\,n\,n[0]+6\,n^2\,n[0]+6\,n[0]^2-6\,n\,n[0]^2+2\,n[0]^3-2\,x[0]-$$
$$2\,n\,y[0]+2\,n[0]\,y[0]+n\,z[0]-n^2\,z[0]-n[0]\,z[0]+2\,n\,n[0]\,z[0]-n[0]^2\,z[0]\}.$$

Further, by initial values substitutions, we derive

$$\{\ -6-6n+z=0,\ -1-3n-3n^2+y,\ x-n^3\ \}.$$

Example 5.29 Given the imperative loop

$$r := 1;\ q := 0;$$
$$\texttt{While}[\ldots,$$
$$\quad t := r;$$
$$\quad r := r+q;$$
$$\quad q := t].$$

By applying Algorithm 5.11 and using `Aligator`, we obtain the polynomial invariant

$$\{\ q^4+2\,q^3\,r-q^2\,r^2-2\,q\,r^3+r^4-q[0]^4-2\,q[0]^3\,r[0]+q[0]^2\,r[0]^2+2\,q[0]\,r[0]^3-r[0]^4\ \},$$

that is

$$\{\ -1+q^4+2\,q^3r-q^2r^2-2\,q\,r^3+r^4\ \},$$

by initial values substitutions.

Example 5.30 Given the imperative loop

$$
\begin{aligned}
&g := 1;\ b := 5/16;\ a := 3/4;\ r := 2;\\
&\texttt{While}[\ldots,\\
&\quad s := t;\\
&\quad t := r;\\
&\quad r := 5*r - 7*a + 4*b;\\
&\quad a := t\\
&\quad b := s;\\
&\quad g := 4*g].
\end{aligned}
$$

By applying Algorithm 5.11 and using `Aligator`, we obtain the polynomial invariant

$$
\begin{aligned}
&\{\ 31\ g\ a[0]^3 - 69\ g\ a[0]^2\ b[0] + 56\ g\ a[0]\ b[0]^2 - 16\ g\ b[0]^3 - 31\ a^3\ g[0] + 69\ a^2\ b\ g[0] -\\
&\ 56\ a\ b^2\ g[0] + 16\ b^3\ g[0] + 32\ a^2\ r\ g[0] - 47\ a\ b\ r\ g[0] + 20\ b^2\ r\ g[0] - 10\ a\ r^2\ g[0] +\\
&\ 7\ b\ r^2\ g[0] + r^3\ g[0] - 32\ g\ a[0]^2\ r[0] + 47\ g\ a[0]\ b[0]\ r[0] - 20\ g\ b[0]^2\ r[0] + 10\ g\ a[0]\ r[0]^2 -\\
&\ 7\ g\ b[0]\ r[0]^2 - g\ r[0]^3\},
\end{aligned}
$$

yielding

$$\{\ 496a^3 + 1104a^2b - 896ab^2 + 256b^3 - g + 512a^2r - 752abr + 320b^2r - 160ar^2 + 112br^2 + 16r^3\ \},$$

by initial values substitutions.

Example 5.31 Given the affine loop

$$
\begin{aligned}
&a := 0;\ b := 0;\\
&\texttt{While}[\ldots,\\
&\quad a := a + 2*b + 1;\\
&\quad b := b + 1].
\end{aligned}
$$

By applying Algorithm 5.11 and using `Aligator`, we obtain the polynomial invariant

$$\{\ a - b^2 - a[0] + b[0]^2\ \},$$

from which, by initial values substitutions, we derive

$$\{\ a - b^2\ \}.$$

Example 5.32 Given the affine loop

$$
\begin{aligned}
&a := 0;\ b := 0; c := 1;\\
&\texttt{While}[\ldots,\\
&\quad a := a + 1;\\
&\quad b := b + c;\\
&\quad c := c + 2].
\end{aligned}
$$

By applying Algorithm 5.11 and using `Aligator`, we obtain the polynomial invariants

$$\{\ 4\,b+2\,c-c^2-4\,b[0]-2\,c[0]+c[0]^2,\ 2\,a-c-2\,a[0]+c[0]\ \},$$

yielding

$$\{\ -1+4b+2c-c^2,\ 1+2a-c\ \},$$

by initial values substitutions.

Further Examples - using `Aligator` *for Imperative Program Verification in* Theorema

In what follows, for the considered examples implementing non-trivial algorithms working on numbers, after illustrating our approach for polynomial invariant generation using `Aligator`, we also present how the automatically generated invariants together with other (user-asserted) non-polynomial invariants can be further used for imperative program verification in *Theorema*.

The annotation process, i.e. integration of `Aligator` in *Theorema* is not yet computer-supported, and is expected to be done manually by the user.

Example 5.33 Given the imperative loop

$$\begin{array}{l} x := 0; y := 0 \\ \texttt{While}[y \neq k, \\ \qquad y := y+1; \\ \qquad x := x+y^5 \\ ]. \end{array}$$

By applying Algorithm 5.11 and using `Aligator`, we obtain the polynomial invariant

$$\{12\,x+y^2-5\,y^4+6\,y^5-2\,y^6-12\,x[0]-y[0]^2+5\,y[0]^4-6\,y[0]^5+2\,y[0]^6\},$$

yielding

$$\{12x+y^2-5y^4+6y^5-2y^6\},$$

by initial value substitutions.

Verification in Theorema

The considered imperative loop is taken from (Petter, 2004) and implements an algorithm for computing the sum of the first n integers at power 5.

The verification of the algorithm, based on the weakest precondition strategy, is supported in the imperative programming environment of *Theorema* (see Section 3). Thus the specification of the algorithm is given below.

$$\begin{array}{l} \texttt{Specification}[\text{"}\textit{Petter}\text{"},\ \textit{GeoPetter}[\downarrow k, \uparrow x], \\ \qquad \texttt{Pre} \rightarrow (k>0), \\ \qquad \texttt{Post} \rightarrow \left(x=\sum_{j=1}^{k} j^5\right)]. \end{array}$$

Using the polynomial invariant generated by `Aligator`, the annotated program code is as follows.

$$\begin{aligned}
&\texttt{Program}[\text{"}\textit{Petter}\text{"},\ \textit{GeoPetter}[\downarrow k, \uparrow x],\\
&\quad \texttt{Module}[\{y\},\\
&\qquad x := 0; y := 0;\\
&\qquad \texttt{WHILE}[(y \neq k),\\
&\qquad\qquad y := y+1;\\
&\qquad\qquad x := x+y^5,\\
&\qquad\qquad \texttt{Invariant} \rightarrow (12x = -y^2+5y^4-6y^5+2y^6)]]].
\end{aligned}$$

Calling the `VCG` for verifying the partial correctness of the "`Petter`" program w.r.t. its given specification, namely:

$$\texttt{VCG}[\texttt{Program}[\text{"}\textit{Petter}\text{"}], \texttt{Specification}[\text{"}\textit{Petter}\text{"}]],$$

we obtain 3 universally quantified verification conditions as a *Theorema* lemma, expressed in first order predicate logic, as follows.

Lemma (Petter) :
for any : k, x, y
(WHILE.Inv)
$(12 * x = -1 * y^2 + 5 * y^4 + (-6) * y^5 + 2 * y^6) \wedge y \neq k \Rightarrow$
$(12 * ((1+y)^5 + x) = -1 * (1+y)^2 + 5 * (1+y)^4 + (-6) * (1+y)^5 + 2 * (1+y)^6)$
(WHILE.Final)

$$(12 * x = -1 * y^2 + 5 * y^4 + (-6) * y^5 + 2 * y^6) \wedge \neg (y \neq k) \Rightarrow \left(x = \sum_{j=1,\ldots,k} (j^5) \right)$$

(Init) $k > 0 \Rightarrow (0 = 0)$

Example 5.34 Given the imperative loop

$$\begin{aligned}
&k := 0;\ j := 1;\ m := 1;\\
&\texttt{While}[(m \leq n),\\
&\qquad k := k+1;\\
&\qquad j := j+2;\\
&\qquad m := m+j].
\end{aligned}$$

By applying Algorithm 5.11 and using `Aligator`, we obtain the polynomial invariants

$$\{\ 2\ j + j^2 - 4\ m - 2\ j[0] - j[0]^2 + 4\ m[0],\ -j + 2\ k + j[0] - 2\ k[0]\ \},$$

yielding

$$\{1 + 2j + j^2 - 4m,\ 1 - j + 2k\},$$

by initial values substitutions.

Verification in Theorema

The considered imperative loop is taken from (Knuth, 1998; Rodriguez-Carbonell and Kapur, 2004) and implements an algorithm for computing the integer square root k of an integer n.

The verification of the algorithm is supported in the imperative programming environment of *Theorema* (see Section 3). Thus the specification of the algorithm is as follows.

$$\begin{aligned}&\texttt{Specification}[\text{"}\textit{IntegerSquareRoot}\text{"}, \textit{IntRoot}[\downarrow n, \uparrow k],\\ &\quad \texttt{Pre} \rightarrow (n \geq 0),\\ &\quad \texttt{Post} \rightarrow ((k^2 \leq n) \ \wedge \ (n \leq (k+1)^2))].\end{aligned}$$

Using the polynomial invariants generated by `Aligator`, as well as other non-polynomial invariant properties, the annotated program code is given below.

$$\begin{aligned}&\texttt{Program}[\text{"}\textit{IntegerSquareRoot}\text{"}, \textit{IntRoot}[\downarrow n, \uparrow k],\\ &\quad \texttt{Module}[\{j,m\},\\ &\qquad k := 0; j := 1; m := 1;\\ &\qquad \texttt{WHILE}[m \leq n,\\ &\qquad\quad k := k+1;\\ &\qquad\quad j := j+2;\\ &\qquad\quad m := m+j,\\ &\qquad\quad \texttt{Invariant} \rightarrow (j = 1+2k) \wedge (4*m = (j+1)^2) \wedge (k^2 \leq n) \wedge (k \geq 0),\\ &\qquad\quad \texttt{TerminationTerm} \rightarrow n-m]].\end{aligned}$$

Calling the `VCG` for verifying the total correctness of the "`Integer Square Root`" program w.r.t. its given specification, namely:

$$\texttt{VCG}[\texttt{Program}[\text{"}\textit{IntegerSquareRoot}\text{"}], \texttt{Specification}[\text{"}\textit{IntegerSquareRoot}\text{"}]],$$

we obtain 4 universally quantified verification conditions as a *Theorema* lemma, expressed in first order predicate logic, presented below.

Lemma (Integer Square Root) :

for any : $\boldsymbol{n}$, $\boldsymbol{k}$, j, m

(WHILE.Inv + Term)

$(j = 1+2*\boldsymbol{k}) \bigwedge (4*m = (1+j)^2) \bigwedge \boldsymbol{k}^2 \leq \boldsymbol{n} \bigwedge \boldsymbol{k} \geq 0 \bigwedge m \leq \boldsymbol{n} \bigwedge (\mathrm{T1} = -1*m+\boldsymbol{n}) \Rightarrow$

$(2+j = 1+2*(1+\boldsymbol{k})) \bigwedge (4*(2+j+m) = (3+j)^2) \bigwedge (1+\boldsymbol{k})^2 \leq \boldsymbol{n} \bigwedge 1+\boldsymbol{k} \geq 0 \bigwedge -2+(-1)*j+(-1)*m+\boldsymbol{n} < \mathrm{T1}$

(WHILE.Final)

$(j = 1+2*\boldsymbol{k}) \bigwedge (4*m = (1+j)^2) \bigwedge \boldsymbol{k}^2 \leq \boldsymbol{n} \bigwedge \boldsymbol{k} \geq 0 \bigwedge m \not\leq \boldsymbol{n} \Rightarrow \boldsymbol{k}^2 \leq \boldsymbol{n} \bigwedge \boldsymbol{n} < (1+\boldsymbol{k})^2$

(WHILE.Term)

$(j = 1+2*\boldsymbol{k}) \bigwedge (4*m = (1+j)^2) \bigwedge \boldsymbol{k}^2 \leq \boldsymbol{n} \bigwedge \boldsymbol{k} \geq 0 \bigwedge m \leq \boldsymbol{n} \Rightarrow -1*m+\boldsymbol{n} \geq 0$

(Init) $\quad \boldsymbol{n} \geq 0 \Rightarrow (1 = 1) \wedge (4 = 4) \wedge 0 \leq \boldsymbol{n}$

Example 5.35 Given the imperative loop

$$\begin{aligned} &x := a/2;\ r := 0;\\ &\texttt{While}[(x > r),\\ &\qquad x := x - r;\\ &\qquad r := r + 1]. \end{aligned}$$

By applying Algorithm 5.11 and using `Aligator`, we obtain the polynomial invariant

$$\{\ -r + r^2 + 2\,x + r[0] - r[0]^2 - 2\,x[0]\ \},$$

yielding

$$\{\ -a - r + r^2 + 2x\ \},$$

by initial values substitutions.

Verification in Theorema

The considered imperative loop is taken from (Kirchner, 1999) and implements another version of an algorithm for computing the integer square root r of an integer a.

The verification of the algorithm is supported in the imperative programming environment of *Theorema* (see Section 3). Thus the specification of the algorithm is as follows.

$$\begin{aligned} &\texttt{Specification}[\text{"}\textit{SquareRoot}\text{"},\ \textit{SquareRoot}[\downarrow a, \uparrow r],\\ &\qquad \texttt{Pre} \rightarrow (a \geq 2),\\ &\qquad \texttt{Post} \rightarrow (r^2 + r + 1 \geq a)\ \wedge\ (r^2 - r - 1 \leq a)]. \end{aligned}$$

Using the polynomial invariant generated by `Aligator`, as well as other non-polynomial invariant properties, the annotated program code is given below.

$$\begin{aligned} &\texttt{Program}[\text{"}\textit{SquareRoot}\text{"},\ \textit{SquareRoot}[\downarrow a, \uparrow r],\\ &\quad \texttt{Module}[\{x\},\\ &\qquad x := a/2; r := 0;\\ &\qquad \texttt{WHILE}[x > r,\\ &\qquad\qquad x := x - r;\\ &\qquad\qquad r := r + 1,\\ &\qquad\qquad \texttt{Invariant} \rightarrow (a + r - r^2 = 2 * x) \wedge (x \geq 1),\\ &\qquad\qquad \texttt{TerminationTerm} \rightarrow x - r]]. \end{aligned}$$

Calling the `VCG` for verifying the total correctness of the "`SquareRoot`" program w.r.t. its given specification, namely:

$$\texttt{VCG}[\texttt{Program}[\text{"}\textit{SquareRoot}\text{"}], \texttt{Specification}[\text{"}\textit{SquareRoot}\text{"}]],$$

we obtain 4 universally quantified verification conditions as a *Theorema* lemma, expressed in first order predicate logic, as follows.

Lemma (SquareRoot) :
for any : a, r, x
(WHILE.Inv + Term)
$(a + r + (-1) * r^2 = 2 * x) \bigwedge x \geq 1 \bigwedge x > r \bigwedge (\mathrm{T1} = x + (-1) * r) \Rightarrow$
$(1 + a + r + (-1) * (1 + r)^2 = 2 * (x + (-1) * r)) \bigwedge x + (-1) * r \geq 1 \bigwedge -1 + x + (-2) * r < \mathrm{T1}$
(WHILE.Final)
$(a + r + (-1) * r^2 = 2 * x) \bigwedge x \geq 1 \bigwedge x \not> r \Rightarrow 1 + r + r^2 \geq a \bigwedge -1 + (-1) * r + r^2 \leq a$
(WHILE.Term)
$(a + r + (-1) * r^2 = 2 * x) \bigwedge x \geq 1 \bigwedge x > r \Rightarrow x + (-1) * r \geq 0$
(Init) $a \geq 2 \Rightarrow (a = a) \bigwedge \frac{1}{2} * a \geq 1$

Example 5.36 Given the affine loop

$$
\begin{aligned}
&x := a;\ r := 1;\ s := 13/4;\\
&\texttt{While}[x - s > 0,\\
&\qquad x := x - s;\\
&\qquad s := s + 6 * r + 3;\\
&\qquad r := r + 1].
\end{aligned}
$$

By applying Algorithm 5.11 and using `Aligator`, we obtain the polynomial invariants

$$
\begin{aligned}
&\{\ -3\ r^2 + s + 3\ r[0]^2 - s[0],\\
&\ r - 3\ r^2 + 2\ r^3 + 2\ x - r[0] + 3\ r[0]^2 - 6\ r\ r[0]^2 + 4\ r[0]^3 + 2\ r\ s[0] - 2\ r[0]\ s[0]\ - 2\ x[0]\ \},
\end{aligned}
$$

yielding

$$\{\ -1 - 12r^2 + 4s,\ -1 - 4a + 3r - 6r^2 + 4r^3 + 4x\ \},$$

by initial values substitutions.

Verification in Theorema

The considered imperative loop is taken from (Rodriguez-Carbonell and Kapur, 2007b) and implements an algorithm for computing the cubic root r for a given integer number a.

The verification of the algorithm is supported in the imperative programming environment of *Theorema* (see Section 3). Thus the specification of the algorithm is as follows.

$$
\begin{aligned}
&\texttt{Specification}[\text{"}\textit{CubicRoot}\text{"},\ \textit{CubicRoot}[\downarrow a, \uparrow r],\\
&\qquad \texttt{Pre} \rightarrow (a \geq 1),\\
&\qquad \texttt{Post} \rightarrow \left(\left(r - \frac{1}{2}\right)^3 < a < \left(r + \frac{1}{2}\right)^3\right)].
\end{aligned}
$$

Using the polynomial invariants generated by `Aligator`, as well as other non-polynomial in-

variant properties, the annotated program code is given below.

$$
\begin{array}{l}
\texttt{Program}[\text{"}\textit{CubicRoot}\text{"},\ \textit{CubicRoot}[\downarrow a, \uparrow r], \\
\quad \texttt{Module}[\{x, s\}, \\
\qquad x := a;\ r := 1;\ s := 13/4; \\
\qquad \texttt{WHILE}[(x - s > 0), \\
\qquad\quad x := x - s; \\
\qquad\quad s := s + 6 * r + 3; \\
\qquad\quad r := r + 1, \\
\qquad \texttt{Invariant} \rightarrow (4 * s = 12 * r^2 + 1) \wedge (4 * x = 1 + 4a - 3r + 6r^2 - 4r^3) \wedge (x \geq 1), \\
\qquad \texttt{TerminationTerm} \rightarrow x - s]]].
\end{array}
$$

Calling the `VCG` for verifying the total correctness of the "`CubicRoot`" program w.r.t. its given specification, namely:

$$\texttt{VCG}[\texttt{Program}[\text{"}\textit{CubicRoot}\text{"}], \texttt{Specification}[\text{"}\textit{CubicRoot}\text{"}]],$$

we obtain 4 universally quantified verification conditions as a *Theorema* lemma, expressed in first order predicate logic, presented below.

Lemma (CubicRoot) :
for any : $\boldsymbol{a}$, $\boldsymbol{r}$, x, s

(WHILE.Inv + Term)

$$(4 * s = 1 + 12 * \boldsymbol{r}^2) \bigwedge (4 * x = 1 + 4 * \boldsymbol{a} + (-3) * \boldsymbol{r} + 6 * \boldsymbol{r}^2 + (-4) * \boldsymbol{r}^3) \bigwedge x \geq 1 \bigwedge -1 * s + x > 0 \bigwedge$$
$$(\mathrm{T1} = -1 * s + x) \Rightarrow$$
$$(4 * (3 + s + 6 * \boldsymbol{r}) = 1 + 12 * (1 + \boldsymbol{r})^2) \bigwedge (4 * (-1 * s + x) = 1 + 4 * \boldsymbol{a} + (-3) * (1 + \boldsymbol{r}) + 6 * (1 + \boldsymbol{r})^2 + (-4) * (1 + \boldsymbol{r})^3) \bigwedge$$
$$-1 * s + x \geq 1 \bigwedge -3 + (-2) * s + x + (-6) * \boldsymbol{r} < \mathrm{T1}$$

(WHILE.Final)

$$(4 * s = 1 + 12 * \boldsymbol{r}^2) \bigwedge (4 * x = 1 + 4 * \boldsymbol{a} + (-3) * \boldsymbol{r} + 6 * \boldsymbol{r}^2 + (-4) * \boldsymbol{r}^3) \bigwedge x \geq 1 \bigwedge -1 * s + x \not> 0 \Rightarrow$$
$$\left(\frac{-1}{2} + \boldsymbol{r}\right)^3 < \boldsymbol{a} < \left(\frac{1}{2} + \boldsymbol{r}\right)^3$$

(WHILE.Term)

$$(4 * s = 1 + 12 * \boldsymbol{r}^2) \bigwedge (4 * x = 1 + 4 * \boldsymbol{a} + (-3) * \boldsymbol{r} + 6 * \boldsymbol{r}^2 + (-4) * \boldsymbol{r}^3) \bigwedge x \geq 1 \bigwedge -1 * s + x > 0 \Rightarrow$$
$$-1 * s + x \geq 0$$

(Init)

$$\boldsymbol{a} \geq 1 \Rightarrow (13 = 13) \wedge (4 * \boldsymbol{a} = 4 * \boldsymbol{a}) \wedge \boldsymbol{a} \geq 1$$

6 P-solvable Loops with Conditionals

Based on results from Section 5, in this section we discuss and present our invariant generation algorithm using algebraic techniques for loops with only conditional branches and assignments.

The starting point of our approach is to do first program transformations (see Theorems 6.1 and 6.6). Namely, transform the P-solvable loop with conditional branches, i.e. outer loops, into nested P-solvable loops with assignments only, i.e. inner loops. Further, we apply steps of Algorithm 5.11 to reason about the inner loops so that at the end we obtain polynomial invariants of the outer loop. Moreover, under additional assumptions introduced in Subsection 6.5, we prove that our approach is complete. Namely, it returns a basis for the polynomial invariant ideal for some special cases of P-solvable loops with conditional branches and assignments. The imposed assumptions in Subsection 6.5 cover a wide class of imperative programs, as shown in Subsection 6.6.

Results presented in this section refine and extend results from (Kovács *et al.*, 2005c; Kovács and Jebelean, 2005; Kovács *et al.*, 2005a; Kovács and Jebelean, 2006; Kovács *et al.*, 2006).

6.1 P-solvable Loops with Conditionals

First, let us show how to eliminate conditionals in the body of an outer loop by replacing them with inner loops.

Theorem 6.1 Consider the following two loops:

$$\texttt{While}[b, \texttt{If}[b_1\ \texttt{Then}\ s_1\ \texttt{Else}\ s_2]] \tag{6.1}$$

and

$$\begin{array}{l}\texttt{While}[b,\\ \quad \texttt{While}[b \wedge b_1, s_1];\\ \quad \texttt{While}[b \wedge \neg b_1, s_2]],\end{array} \tag{6.2}$$

where $b,\ b_1$ are boolean expressions and $s_1,\ s_2$ are sequences of assignments.

Then any formula I is an invariant of the first loop if and only if it is an invariant of the second loop and of both its inner loops.

Proof. We rely on Hoare triple notation and the weakest precondition calculus (Floyd, 1967; Dijkstra, 1976), presented in Subsection 2.3.

I is a loop invariant for (6.1) if and only if

$$\{I \wedge b\}\texttt{If}[b_1\ \texttt{Then}\ s_1\ \texttt{Else}\ s_2]\{I\}. \tag{6.3}$$

Using the wp rules for conditionals, (6.3) holds if and only if the verification condition

$$I \wedge b \implies \big(b_1 \implies \mathrm{wp}(s_1,\, I)\big) \bigwedge \big(\neg b_1 \implies \mathrm{wp}(s_2,\, I)\big)$$

holds, that is equivalent to the following two formulas:

$$I \wedge b \wedge b_1 \implies \mathrm{wp}(s_1,\, I); \tag{6.4}$$

$$I \wedge b \wedge \neg b_1 \implies \mathrm{wp}(s_2,\, I). \tag{6.5}$$

Thus I is a loop invariant of (6.1) if and only if formulas (6.4) and (6.5) hold. Note that these two formulas express that I is an invariant of the two inner loops of (6.2).

Also note that loops (6.1) and (6.2) are equivalent, hence they have the same invariants. The equivalence can be proved by showing that they have the same execution paths.

Now, assume that I is an invariant of (6.1). Since (6.1) and (6.2) have the same invariants, then I is also an invariant of (6.2), and by what we proved before, also the invariant of the inner loops of (6.2).

Conversely, suppose I is an invariant of (6.2) and of its inner loops. Since (6.1) and (6.2) have the same invariants, then I is also an invariant of (6.1).

■

Example 6.2 Consider the imperative loop given below.

$$\begin{array}{l} \texttt{While}[(d \geq Tol), \\ \quad \texttt{If}[(P < a+b) \\ \qquad \texttt{Then}\ b := b/2; d := d/2 \\ \qquad \texttt{Else}\ a := a+b; y := y+d/2; b := b/2; d := d/2]]. \end{array} \tag{6.6}$$

By applying Theorem 6.1, the obtained nested loop system is as follows.

$$\begin{array}{l} \texttt{While}[(d \geq Tol), \\ \quad \texttt{While}[(d \geq Tol) \wedge (P < a+b), \\ \qquad b := b/2; d := d/2]; \\ \quad \texttt{While}[(d \geq Tol) \wedge \neg(P < a+b), \\ \qquad a := a+b; y := y+d/2; b := b/2; d := d/2]]. \end{array} \tag{6.7}$$

Theorem 6.3 Let $s_0,\ s_1,\ s_2,\ s_3$ be sequences of assignments, b a boolean expression and $b' = \mathrm{wp}(s_0,\, b)$. Then

$$\{P\}\ s_0;\ \texttt{If}[b\ \texttt{Then}\ s_1\ \texttt{Else}\ s_2];\ s_3\ \{Q\} \tag{6.8}$$

if and only if

$$\{P\}\ \texttt{If}[b'\ \texttt{Then}\ s_0; s_1; s_3\ \texttt{Else}\ s_0; s_2; s_3]\ \{Q\}. \tag{6.9}$$

Proof. We rely on the weakest precondition calculus (see Subsection 2.3).

Thus, (6.8) holds if and only if

$$P \implies \mathrm{wp}(s_0,\ \texttt{If}[b\ \texttt{Then}\ s_1 \texttt{Else}\ s_2]; s_3,\ Q),$$

which, using the wp rules for sequencing and conditionals, as well as the fact that (in case of assignments) wp commutes with logical connectives, yields the formula

$$P \implies \big(b' \implies \mathrm{wp}(s_0;s_1;s_3,\ Q)\big) \wedge \big(\neg b' \implies wp(s_0;s_2;s_3,\ Q)\big). \tag{6.10}$$

Similarly, (6.9) holds if and only if

$$P \implies \mathrm{wp}(\texttt{If}[b' \texttt{ Then } s_0;s_1;s_3 \texttt{ Else } s_0;s_2;s_3],\ Q), \tag{6.11}$$

which, using the wp rules for conditionals, yields the formula

$$P \implies \big(b' \implies \mathrm{wp}(s_0;s_1;s_3,\ Q)\big) \wedge \big(\neg b' \implies wp(s_0;s_2;s_3,\ Q)\big). \tag{6.12}$$

Thus, proving Theorem 6.3 is straightforward. Indeed,

- (6.8) holds if and only if (6.10) is true, so (6.12), hence (6.9) is valid;
- (6.9) holds if and only if (6.12) is true, so (6.10), hence (6.8) is valid.

■

Based on Theorems 6.1 and 6.3, we have the following result.

Corollary 6.4 Let us consider the following two loops:

$$\texttt{While}[b, s_0; \texttt{If}[b_1 \texttt{ Then } s_1 \texttt{ Else } s_2]; s_3] \tag{6.13}$$

and

$$\begin{array}{l} \texttt{While}[b, \\ \quad \texttt{While}[b \wedge b'_1, s_0;s_1;s_3]; \\ \quad \texttt{While}[b \wedge \neg b'_1, s_0;s_2;s_3]], \end{array} \tag{6.14}$$

where b, b_1 are boolean expressions, s_0, s_1, s_2, s_3 are sequences of assignments, and $b'_1 = \mathrm{wp}(s_0, b_1)$.

Then any formula I is an invariant of the first loop if and only if it is an invariant of the second loop and of both its inner loops.

Proof. It follows immediately from Theorems 6.3 and 6.1.

■

In the transformation rule given by Theorem 6.3, all variables in condition b of the conditional statement from (6.8) which possibly change due to the assignments s_0 before the conditional statement must be suitably modified in the condition of conditional statement (6.9). This is performed by computing the weakest precondition $\mathrm{wp}(s_0, b)$.

Example 6.5 Consider the following loop with conditional and assignment statements.

$$\begin{array}{l}\texttt{While}[b \neq B,\\ \quad q := 2*q;\ b := b/2;\\ \quad \texttt{If}[r \geq b\ \texttt{Then}\ q := q+1;\ r := r-b]].\end{array}$$

By applying Corollary 6.4, we obtain the nested loop system presented below.

$$\begin{array}{l}\texttt{While}[(b \neq B),\\ \quad \texttt{While}[(b \neq B)\ \wedge\ (r \geq b/2),\\ \quad\quad q := 2*q;\ b := b/2;\ q := q+1;\ r := r-b];\\ \quad \texttt{While}[(b \neq B)\ \wedge\ \neg(r \geq b/2),\\ \quad\quad q := 2*q;\ b := b/2]],\end{array}$$

where the condition $r \geq b$ from the conditional statement is suitably modified into $r \geq b/2$ as a part of the conditions of the inner loops, by taking into consideration the effect of the assignment $b := b/2$ before the conditional statement.

Theorem 6.1 (and thus Corollary 6.4) can be generalized to the case when the loop body contains a nested conditional statement with $k \geq 1$ *conditional branches, each conditional branch containing a sequence of assignments*. In this case, by applying recursively the transformation rule from Theorem 6.1, we obtain an outer loop with k inner loops, each inner loop containing only assignment statements, as it is stated below.

Theorem 6.6 Let us consider the following two loops:

$$\begin{array}{l}\texttt{While}[b,\\ \quad \texttt{If}[b_1\ \texttt{Then}\ s_1\\ \quad \texttt{Else If}[b_2\ \texttt{Then}\ s_2\\ \quad \texttt{Else If}[b_3\ \texttt{Then}\ s_3\\ \quad \vdots\\ \quad \texttt{Else If}[b_{k-1}\ \texttt{Then}\ s_{k-1}\\ \quad \texttt{Else}\ s_k]]\end{array} \tag{6.15}$$

and

$$\begin{array}{l}\texttt{While}[b,\\ \texttt{While}[b \wedge b_1, s_1];\\ \texttt{While}[b \wedge \neg b_1 \wedge b_2, s_2];\\ \ldots\\ \texttt{While}[b \wedge \neg b_1 \wedge \cdots \wedge \neg b_{k-1}, s_k]],\end{array} \tag{6.16}$$

where b, b_1, …, b_{k-1} are boolean expressions and s_1, …, s_k are sequences of assignments.
Then any formula I is an invariant of the first loop if and only if it is an invariant of the second loop and all of its inner loops.

Proof. The proof is similar to the proof of Theorem 6.1, using the weakest precondition calculus for sequencing, conditionals and loops.

■

Similarly to Corollary 6.4, based on Theorems 6.6 and 6.3, we have the corollary below.

Corollary 6.7 Let us consider the following two loops:

$$\begin{array}{l}\texttt{While}[b,\\ \quad s_0;\\ \quad \texttt{If}[b_1\ \texttt{Then}\ s_1\\ \quad \texttt{Else If}[b_2\ \texttt{Then}\ s_2\\ \quad \texttt{Else If}[b_3\ \texttt{Then}\ s_3\\ \quad \vdots\\ \quad \texttt{Else If}[b_{k-1}\ \texttt{Then}\ s_{k-1}\\ \quad \texttt{Else}\ s_k];\\ \quad s_{k+1}]\end{array} \tag{6.17}$$

and

$$\begin{array}{l}\texttt{While}[b,\\ \texttt{While}[b\wedge b'_1,\ s_0;s_1;s_{k+1}];\\ \texttt{While}[b\wedge\neg b'_1\wedge b'_2,\ s_0;s_2;s_{k+1}];\\ \ldots\\ \texttt{While}[b\wedge\neg b'_1\wedge\cdots\wedge\neg b'_{k-1},\ s_0;s_k;s_{k+1}]],\end{array} \tag{6.18}$$

where $b,\ b_1,\ \ldots, b_{k-1}$ are boolean expressions, $s_0,\ s_1,\ \ldots,\ s_k,\ s_{k+1}$ are sequences of assignments, and $b'_i = \mathrm{wp}(s_0,\ b_i),\ i = 1,\ldots,k-1$.

Then any formula I is an invariant of the first loop if and only if it is an invariant of the second loop and all of its inner loops.

Proof. It obviously follows from Theorems 6.6 and 6.3.

■

Based on Theorem 6.6 and Definition 5.9, we can now define the notion of P-solvable loops with conditional branches and assignments.

Definition 6.8 An imperative loop having $k \geq 1$ *conditional branches and assignment statements only* is called *P-solvable* if the inner loops obtained after performing the transformation rule from Theorem 6.6 are P-solvable.

Let us introduce some notation. Denote by S_i, where $i = 1,\ldots,k$, the sequence of assignments $s_0;s_i;s_{k+1}$. Using this notation, (6.18) becomes

$$\begin{array}{l}\texttt{While}[b,\\ \texttt{While}[b\wedge b'_1,\ S_1];\\ \texttt{While}[b\wedge\neg b'_1\wedge b'_2,\ S_2];\\ \ldots\\ \texttt{While}[b\wedge\neg b'_1\wedge\cdots\wedge\neg b'_{k-1},\ S_k]].\end{array} \tag{6.19}$$

Similarly to the previously considered P-solvable loops with only assignments, we will *drop the loop condition and tests of the conditional branches*, this turning our loop body into a non-deterministic program fragment. Ignoring the tests in the conditional branches means that either branch is executed in every possible way, whereas ignoring the test condition of the loop means the loop is executed arbitrarily many nonzero times. We will refer to the loop obtained in this way by dropping the loop condition and all test conditions also as a *P-solvable loop.* In the rest of this thesis we will focus on *non-deterministic P-solvable loops with assignments and conditional branches with ignored conditions,* i.e. *conditional branches with ignored tests are treated as non-deterministic branches*.

Using notation introduced on page 46 the considered P-solvable loops with assignments and conditional branches, in which all tests are ignored, can be written as follows:

$$\begin{array}{l} \texttt{While}[\ldots, \\ \quad \texttt{If}[\ldots\ \texttt{Then}\ S_1] \\ \quad \texttt{[]} \\ \quad \vdots \\ \quad \texttt{[]} \\ \quad \texttt{If}[\ldots\ \texttt{Then}\ S_k]]. \end{array} \tag{6.20}$$

Example 6.9 The assignments statements of the inner loops from (6.7) of Example 6.2 are affine. Based on Theorem 5.20, the inner loops are thus P-solvable, yielding that (6.6) is a P-solvable loop. By dropping the test conditions and using the above introduced notation, (6.6) can be written as below.

$$\begin{array}{ll} & \texttt{While}[\ldots, \\ S_1: & \quad \texttt{If}[\ldots\ \texttt{Then}\ b := b/2; d := d/2] \\ & \quad \texttt{[]} \\ S_2: & \quad \texttt{If}[\ldots\ \texttt{Then}\ a := a+b; y := y+d/2; b := b/2; d := d/2]], \end{array}$$

yielding thus a nested loop system with two inner loops S_1 and S_2 as presented in (6.7). Namely, we have

$$\begin{array}{ll} & \texttt{While}[\ldots, \\ S_1: & \texttt{While}[\ldots,\ b := b/2; d := d/2] \\ & \texttt{[]} \\ S_2: & \texttt{While}[\ldots,\ a := a+b; y := y+d/2; b := b/2; d := d/2]]. \end{array}$$

Since tests are ignored in the loop and conditional branches, using our notation for basic non-deterministic programs from Subsection 5.1, the outer loop can be written as

$$(S_1|S_2|\ldots|S_k)^*.$$

Remark 6.10 Note that $(S_1|\ldots|S_k)^*$ is equivalent to $(S_1^*|\ldots|S_k^*)^*$.

We have now all "ingredients" to define the notion of *polynomial invariants among the loop variables X with initial values X_0 for P-solvable loops with conditional branches and assignments as in* (6.20).

Definition 6.11 A polynomial $p \in \mathbb{K}[X]$ is a polynomial invariant of (6.20) among the loop variables X with initial values X_0 if the Hoare triple

$$\{p(X) = 0 \wedge X = X_0\} \quad (S_1|\ldots|S_k)^* \quad \{p(X) = 0\} \tag{6.21}$$

holds.

If (6.21) is valid, we also say that $p(X)$ is a *polynomial invariant among the loop variables X parameterized by X_0*.

Similarly to Section 5, $p \in \mathbb{K}[X]$ is thus a polynomial invariant of (6.20) among the loop variables X with initial values X_0 if and only if the conditions below are satisfied.

(1) $p(X)$ is valid for the initial values of the loop variables. Namely, $p(X)$ is 0 whenever X are evaluated at X_0. Thus, $p(X)$ *holds at the entry/beginning of the loop*.

(2) $p(X)$ is valid after arbitrary many execution of the P-solvable loop (6.20) with non-deterministic branches and assignments, i.e. after arbitrary many execution of $S_1, \ldots, S_k$ in any order, starting in a state in which the initial values of the loop variables X are X_0. Thus, $p(X)$ is *preserved by any execution of the P-solvable loop* (6.20).

Note that Definition 6.11 generalizes Definition 5.5, since in the case of a P-solvable loop with assignments only we have $k = 1$.

Further, Definition 5.8 remains unchanged: a *polynomial loop invariant of* (6.20) is a conjunction of polynomial invariant equations of (6.20) among the loop variables X with initial values X_0.

Hence, having a P-solvable loop with conditional branches, after applying the transformation rule from Theorem 6.6 (or Corollary 6.7), we can apply Algorithm 5.11 to reason about the P-solvable inner loops and determine their closed form systems, in order to obtain polynomial invariants of the outer P-solvable loop (see Algorithm 6.32).

Moreover, under additional assumptions (see Subsection 6.5), we prove that our approach is complete. Namely, it generates a basis for the polynomial invariant ideal for some special cases of P-solvable loops with conditional branches and assignment statements. However, we could not find any example of a P-solvable loop with conditional branches and assignments for which our method fails to be complete. Thus we conjecture that the imposed restrictions from Subsection 6.5 handle a rich class of imperative loops. For all examples treated in Subsection 6.6, our approach returns a basis of the polynomial invariant ideal.

The study of completeness of our approach without the additional assumptions of Subsection 6.5 remains a challenging task for further research.

In what follows, we consider a generalization of Algorithm 5.11. Namely, we present our approach for invariant generation for P-solvable loops with conditional branches and assignments.

The main steps of the algorithm are as follows.

(1) Firstly, we perform a simple program transformation (see Theorem 6.6). Namely, we transform the outer P-solvable loop with conditional branches (non-deterministic branches) into nested P-solvable loops with assignments only (i.e. inner loops), and then apply Algorithm 5.11 to get the closed form solution system of each inner loop.

(2) Further, by "merging" the closed form solution systems (see Definitions 6.15 and 6.16) and using the fact that the initial values e.g. of the second inner loop are given by the final values of the first inner loop, and eliminating the variables in the inner loop indexes, we obtain all polynomial relations for an *arbitrary iteration* of the outer loop (see Theorem 6.23). That is the ideal of all polynomial relations after any sequence of k P-solvable inner loops describing a possible iteration of the outer loop.

(3) Finally, from the ideal of all polynomial relations after any sequence of k P-solvable inner loops describing a possible *first* iteration of the outer loop, we keep only those polynomials that are preserved on each conditional branch.

In this way, we obtain *polynomial invariants* for the P-solvable loop with conditional branches and assignments (see Theorem 6.34).

Moreover, under *additional assumptions*, in Subsection 6.5 we show that the set of polynomial invariants thus obtained forms a *basis for the polynomial invariant ideal* of the P-solvable loop with conditional branches and assignments. Hence, Subsection 6.5 shows under which assumptions our approach is complete in the sense that it generates a basis of the polynomial invariant ideal from which any other polynomial invariant of the P-solvable loop with conditional branches and assignments can be derived.

The steps of our invariant generation algorithm for P-solvable loops with conditional branches and assignments are presented in details below.

6.2 Merging of Closed Form Systems for Inner Loop Sequences

Each inner loop in (6.19) is P-solvable. The closed form system of its variables is given thus as in (5.4), namely:

$$\mathrm{CF}(S_i^{j_i},\, E_{S_i}, X_i, X_{0i}):\quad \begin{cases} x_1[j_i] = & p_{i,1,1}(j_i)\theta_{i1}^{j_i} + \cdots + p_{i,1,s}(j_i)\theta_{is}^{j_i} \\ x_2[j_i] = & p_{i,2,1}(j_i)\theta_{i1}^{j_i} + \cdots + p_{i,2,s}(j_i)\theta_{is}^{j_i} \\ \vdots & \\ x_m[j_i] = & p_{i,m,1}(j_i)\theta_{i1}^{j_i} + \cdots + p_{i,m,s}(j_i)\theta_{is}^{j_i}, \end{cases} \tag{6.22}$$

where

(1) $X_i = \{x_1[j_i], \ldots, x_m[j_i]\}$ represents the values of loop variables $X = \{x_1, \ldots, x_m\}$ after j_i iterations of the ith inner loop (i.e. $S_i^{j_i}$);

(2) X_{0i} represents the initial values of loop variables $X = \{x_1, \ldots, x_m\}$ before entering the ith inner loop (i.e. $j_i = 0$);

(3) $p_{i,l,r} \in \bar{\mathbb{K}}[j_i]$, $l = 1, \ldots, m, r = 1, \ldots, s$. The coefficients of $p_{i,l,r}$ are given by the initial values X_{0i} of the loop variables before entering the ith inner loop;

(4) $E_{S_i} = \{\theta_{i1}^{j_i}, \ldots, \theta_{is}^{j_i}\}$;

(5) there exist algebraic dependencies among $\theta_{i1}^{j_i},\dots,\theta_{is}^{j_i}$. The ideal of algebraic dependencies of $\theta_1^{j_i},\dots,\theta_s^{j_i}$ is computed by Algorithm 4.45, and we denote

$$A_i := I(\theta_{i1}^{j_i},\dots,\theta_{is}^{j_i}) = I(j_i,\theta_{i1}^{j_i},\dots,\theta_{is}^{j_i}) \trianglelefteq \bar{\mathbb{K}}[z_{i0},z_{i1},\dots,z_{is}].$$

W.l.o.g. we assumed that the number s of exponential sequences in the closed form system (6.22) is the same for every $i=1,\dots,k$, since

- we can always add terms $0\cdot\theta_r^{j_i}$,
- by renaming, we can have the same number of exponential terms in each closed form.

Moreover, since $\theta_{i1}^{j_i},\dots,\theta_{is}^{j_i}$ are polynomially related (i.e. $A_i\neq\emptyset$) and the coefficients of $p_{i,l,r}$ are given by the initial values X_{0i}, we can rewrite (6.22) as a polynomial system given below.

$$\mathrm{CF}(S_i^{j_i},\ E_{S_i},X_i,X_{0i}):\quad \begin{cases} x_1[j_i] = & q_{i,1}(j_i,\theta_{i1}^{j_i},\dots,\theta_{is}^{j_i}) \\ x_2[j_i] = & q_{i,2}(j_i,\theta_{i1}^{j_i},\dots,\theta_{is}^{j_i}) \\ \vdots & \\ x_m[j_i] = & q_{i,m}(j_i,\theta_{i1}^{j_i},\dots,\theta_{is}^{j_i}), \end{cases} \tag{6.23}$$

where $q_{i,l}\in\bar{\mathbb{K}}[z_{i0},z_{i1},\dots,z_{is}]$, $l=1,\dots,m$, and the coefficients of $q_{i,l}$ are given by the initial values X_{0i}. The variables $z_{i0},z_{i1},\dots,z_{is}$ stand for $j_i,\theta_{i1}^{j_i},\dots,\theta_{is}^{j_i}$.

Example 6.12

The closed form systems of the P-solvable inner loops S_1 and S_2 from Example 6.9 (or equivalently, from Example 6.2) are computed using Algorithm 5.11, as presented below.

Extracting system of recurrences for each inner loop.

Inner loop S_1:

$j_1\in\mathbb{N}$

$$\begin{cases} a[j_1+1] = & a[j_1] \\ b[j_1+1] = & b[j_1]/2 \\ d[j_1+1] = & d[j_1]/2 \\ y[j_1+1] = & y[j_1] \end{cases}$$

Inner loop S_2:

$j_2\in\mathbb{N}$

$$\begin{cases} a[j_2+1] = & a[j_2]+b[j_2] \\ y[j_2+1] = & y[j_2]+d[j_2]/2 \\ b[j_2+1] = & b[j_2]/2 \\ d[j_2+1] = & d[j_2]/2, \end{cases}$$

where j_1 and j_2 represent respectively the loop counters of the inner loops S_1 and S_2.

Closed form systems of inner loops.

The closed forms of the inner loops S_1 and S_2 are finally obtained by

(1) solving their system of recurrences,

(2) computing the algebraic dependencies of the exponential sequences from the closed forms of the loop variables.

The result is listed below.

Inner loop S_1:
$j_1 \in \mathbb{N}$
$z_{11} = 2^{-j_1}, z_{12} = 2^{-j_1}$

$\mathrm{CF}(S_1^{j_1},\ E_{S_1}, X_1, X_{01})$:

$$\begin{cases} a[j_1] & = & a[0_1] \\ b[j_1] & \underset{C-finite}{=} & b[0_1] * z_{11} \\ d[j_1] & \underset{C-finite}{=} & d[0_1] * z_{12} \\ y[j_1] & = & y[0_1] \end{cases}$$

Inner loop S_2:
$j_2 \in \mathbb{N}$
$z_{21} = 2^{-j_2}, z_{22} = 2^{-j_2}, z_{23} = 2^{-j_2}, z_{24} = 2^{-j_2}$

$\mathrm{CF}(S_2^{j_2},\ E_{S_2}, X_2, X_{02})$:

$$\begin{cases} a[j_2] & \underset{Gosper}{=} & a[0_2] + 2 * b[0_2] - 2 * b[0_2] * z_{21} \\ b[j_2] & \underset{C-finite}{=} & b[0_2] * z_{22} \\ d[j_2] & \underset{C-finite}{=} & d[0_2] * z_{23} \\ y[j_2] & \underset{Gosper}{=} & y[0_2] + d[0_2] - d[0_2] * z_{24}, \end{cases}$$

with the computed algebraic dependencies

$$\{\ z_{11} - z_{12} = 0 \qquad \text{and} \qquad \begin{cases} z_{21} - z_{24} = 0 \\ z_{22} - z_{24} = 0 \\ z_{23} - z_{24} = 0, \end{cases}$$

where

- $X_{01} = \{a[0_1], b[0_1], d[0_1], y[0_1]\}$ and $X_{02} = \{a[0_2], b[0_2], d[0_2], y[0_2]\}$ are respectively the initial values of a, b, d, y before entering the inner loops S_1 and S_2;
- z_{11} and z_{12} stand for the exponential sequences $E_{S_1} = \{2^{-j_1}, 2^{-j_1}\}$ from the closed form system of the inner loop S_1;
- z_{21}, z_{22}, z_{23} and z_{24} stand for the exponential sequences $E_{S_2} = \{2^{-j_2}, 2^{-j_2}, 2^{-j_2}, 2^{-j_2}\}$ from the closed form system of the inner loop S_2.

Remark 6.13 Similarly to Remark 6.10, $(S_1|\ldots|S_k)^*$ is equivalent to $(S_1^* * \cdots * S_k^*)^*$. We will use this equivalence and switch to studying permutations and iterations of inner loops to describe the behavior of $S_1^* * \cdots * S_k^*$ as follows.

In the general case of a P-solvable loop with a nested conditional statement having $k \geq 1$ conditional branches, by applying Theorem 6.6, we obtain an outer loop with k P-solvable inner loops $S_1, \ldots, S_k$ with only assignments (i.e. the loop body S_i is a sequence of assignments). Thus an *arbitrary iteration* of the outer loop is described by an arbitrary sequence of the k P-solvable loops. Since the tests are ignored (yielding this way non-deterministic branches in the loop body) and hence the inner loops can be executed in an arbitrary order, for any iteration of the outer loop we have $k!$ possible sequences of inner P-solvable loops.

For simplicity of further notation, let us denote the set of permutations of length k over $\{1, \ldots, k\}$ by $\mathfrak{S}_k$.

Consider a permutation $W = (w_1, \ldots, w_k) \in \mathfrak{S}_k$ and a sequence of numbers $J = \{j_1, \ldots, j_k\} \in \mathbb{N}^k$. Then we write $S_W^J = S_{w_1}^{j_1}; S_{w_2}^{j_2}; \ldots; S_{w_k}^{j_k}$ to denote an *arbitrary iteration of the outer loop*, i.e. an arbitrary sequence of the k inner loops.

Thus, we can now characterize all possible iterations of the nested outer loop with k P-solvable inner loops as being the set

$$IS_k = \{S_W^J \mid W \in \mathfrak{S}_k, J = \{j_1, \ldots, j_k\} \in \mathbb{N}^k\}. \tag{6.24}$$

In the case of (6.1) (or equivalently (6.14)), we have $k = 2$ and

$$IS_2 = \{S_1^{j_1}; S_2^{j_2}, \quad S_2^{i_2}; S_1^{i_1} \mid j_1,\ j_2, i_1, i_2 \in \mathbb{N}\}.$$

We have already defined the closed form system for a single inner loop S_i with j_i iterations as $S_i^{j_i}$, see (6.22). Now we would like to define the *merged closed form system* of the loop variables after an arbitrary sequence $S_{w_1}, \ldots, S_{w_u}$ of u P-solvable inner loops with $w_1, \ldots, w_u \in \{1, \ldots, k\}$, $u \geq 2$ and $j_1, \ldots, j_u \in \mathbb{N}$ iterations respectively.

First, let us state the following theorem.

Theorem 6.14 Let $u \geq 1$, $w_1, \ldots, w_u \in \{1, \ldots, k\}$ and $j_1, \ldots, j_u \in \mathbb{N}$.

The closed form of each loop variable from $X = \{x_1, \ldots, x_m\}$ after an arbitrary sequence $S_{w_1}, \ldots, S_{w_u}$ of u P-solvable inner loops with respectively $j_1, \ldots, j_u \in \mathbb{N}$ iterations is a polynomial in the loop counters $j_1, \ldots, j_u$ and polynomially related exponential sequences in $j_1, \ldots, j_u$.

The coefficients of these polynomials are given by the initial values X_{0w_1} of loop variables before the arbitrary sequence $S_{w_1}, \ldots, S_{w_u}$ with respectively $j_1, \ldots, j_u \in \mathbb{N}$ iterations, i.e. before $S_{w_1}^{j_1}$ (cf. notation from (6.22)).

Thus, the closed form system of loop variables after $S_{w_1}^{j_1}, \ldots, S_{w_u}^{j_u}$ is

$$\begin{cases} x_1[j_1, \ldots, j_u] = & q_1(j_1, \theta_{w_1 1}^{j_1}, \ldots, \theta_{w_1 s}^{j_1}, \ldots \ldots, j_u, \theta_{w_u 1}^{j_u}, \ldots, \theta_{w_u s}^{j_u}) \\ \vdots & \\ x_m[j_1, \ldots, j_u] = & q_m(j_1, \theta_{w_1 1}^{j_1}, \ldots, \theta_{w_1 s}^{j_1}, \ldots \ldots, j_u, \theta_{w_u 1}^{j_u}, \ldots, \theta_{w_u s}^{j_u}), \end{cases} \tag{6.25}$$

where

- $x_l[j_1, \ldots, j_u]$, $l = 1, \ldots, m$, represent the values of the loop variables x_l after respectively $j_1, \ldots, j_u$ iterations of $S_{w_1}, \ldots, S_{w_u}$, i.e. after $\underbrace{S_{w_1}; \ldots; S_{w_1}}_{j_1 \text{ times}}; \ldots; \underbrace{S_{w_u}; \ldots; S_{w_u}}_{j_u \text{ times}}$;
- $q_l \in \bar{\mathbb{K}}[z_{10}, z_{11}, \ldots, z_{1s}, \ldots \ldots, z_{u0}, z_{u1}, \ldots, z_{us}]$, $l = 1 \ldots, m$, and the coefficients of q_l are given by the initial values X_{0w_1}.

The variables $z_{i0}, z_{i1}, \ldots, z_{is}$ stand for the polynomially related sequences $j_i, \theta_{w_i 1}^{j_i}, \ldots, \theta_{w_i s}^{j_i}$, $i = 1, \ldots, u$.

Proof. We prove by induction over the length $u \geq 1$ of the sequence of inner loops.

Induction base: $u = 1$.

There is nothing to prove, since from (6.23) we already have the sought closed form for the loop variables after j_1 iterations of S_{w_1} as in (6.25).

Induction step.

$\forall 1 \leq v \leq u-1$, we assume that the closed form system of the loop variables $X = \{x_1, \ldots, x_m\}$ after an arbitrary sequence $S_{w_1}, \ldots, S_{w_v}$ of v P-solvable inner loops with respectively $j_1, \ldots, j_v$ iterations is as (6.25). Namely, we have

$$\begin{cases} x_1[j_1, \ldots, j_v] = & q_{v,1}(j_1, \theta_{w_1 1}^{j_1}, \ldots, \theta_{w_1 s}^{j_1}, \ldots \ldots, j_v, \theta_{w_v 1}^{j_v}, \ldots, \theta_{w_v s}^{j_v}) \\ \vdots & \\ x_m[j_1, \ldots, j_v] = & q_{v,m}(j_1, \theta_{w_1 1}^{j_1}, \ldots, \theta_{w_1 s}^{j_1}, \ldots \ldots, j_v, \theta_{w_v 1}^{j_v}, \ldots, \theta_{w_v s}^{j_v}), \end{cases} \tag{6.26}$$

where

- $q_{v,l} \in \bar{\mathbb{K}}[z_{10}, z_{11}, \ldots, z_{1s}, \ldots \ldots, z_{v0}, z_{v1}, \ldots, z_{vs}]$, $l = 1 \ldots, m$, with coefficients being determined by the initial values X_{0w_1} of the loop variables $X = \{x_1, \ldots, x_m\}$ before $S_{w_1}^{j_1}$;
- the sequences $\theta_{w_i 1}^{j_i}, \ldots, \theta_{w_i s}^{j_i}$, $i = 1, \ldots, v$, are polynomially related;
- the variables $z_{i0}, z_{i1}, \ldots, z_{is}$ stand for $j_i, \theta_{w_i 1}^{j_i}, \ldots, \theta_{w_i s}^{j_i}$, $i = 1, \ldots, v$.

By (6.22), the closed form of the loop variables $X = \{x_1, \ldots, x_m\}$ after j_u iterations of S_{w_u} is

$$\mathrm{CF}(S_{w_u}^{j_u},\ E_{S_{w_u}}, X_{w_u}, X_{0w_u}) : \quad \begin{cases} x_1[j_u] = & q_{u,1}(j_u, \theta_{w_u 1}^{j_u}, \ldots, \theta_{w_u s}^{j_u}) \\ \vdots & \\ x_m[j_u] = & q_{u,m}(j_u, \theta_{w_u 1}^{j_u}, \ldots, \theta_{w_u s}^{j_u}), \end{cases} \tag{6.27}$$

where

- $q_{u,l} \in \bar{\mathbb{K}}[z_{u0}, z_{u1}, \ldots, z_{us}]$, $l = 1 \ldots, m$;
- the coefficients of $q_{u,l}$ are determined by the initial values X_{0w_u} of the loop variables $X = \{x_1, \ldots, x_m\}$ before $S_u^{j_u}$;
- the sequences $\theta_{w_u 1}^{j_u}, \ldots, \theta_{w_u s}^{j_u}$ are polynomially related;
- the variables $z_{u0}, z_{u1}, \ldots, z_{us}$ stand for $j_u, \theta_{w_u 1}^{j_u}, \ldots, \theta_{w_u s}^{j_u}$.

Further, considering the sequence $S_{w_1}^{j_1}; \ldots; S_{w_{u-1}}^{j_{u-1}}; S_{w_u}^{j_u}$ of u inner loops, note that the initial values X_{0w_u} of the loop variables before $S_{w_u}^{j_u}$ are the values of the loop variables after $S_{w_1}^{j_1}; \ldots; S_{w_{u-1}}^{j_{u-1}}$, i.e. $x_l[j_1, \ldots, j_{u-1}]$, $l = 1, \ldots, m$, respectively.

Thus, based on the induction hypothesis (6.26), the initial values X_{0w_u} are polynomials in the loop counters $j_1, \ldots, j_{u-1}$ and polynomially related exponential sequences in $j_1, \ldots, j_{u-1}$, with coefficients being determined by the initial values X_{0w_1}. In other words, each initial value from X_{0u}

is a polynomial $q_{u-1,l}(j_1, \theta^{j_1}_{w_1 1}, \ldots, \theta^{j_s}_{w_1 s}, \ldots\ \ldots, j_{u-1}, \theta^{j_{u-1}}_{w_{u-1} 1}, \ldots, \theta^{j_{u-1}}_{w_{u-1} s})$ over $\bar{\mathbb{K}}$, with coefficients being determined by the initial values X_{0w_1}.

Since composition of polynomials yields a polynomial expression, based on (6.27), the closed forms of the loop variables after $S^{j_1}_{w_1}; \ldots; S^{j_u}_{w_u}$ is as (6.25). Namely, we have

$$\begin{cases} x_1[j_1, \ldots, j_u] = & f_1(j_1, \theta^{j_1}_{w_1 1}, \ldots, \theta^{j_1}_{w_1 s}, \ldots\ \ldots, j_u, \theta^{j_u}_{w_u 1}, \ldots, \theta^{j_u}_{w_u s}) \\ \vdots & \\ x_m[j_1, \ldots, j_u] = & f_m(j_1, \theta^{j_1}_{w_1 1}, \ldots, \theta^{j_1}_{w_1 s}, \ldots\ \ldots, j_u, \theta^{j_u}_{w_u 1}, \ldots, \theta^{j_u}_{w_u s}), \end{cases} \tag{6.28}$$

where $f_l \in \bar{\mathbb{K}}[z_{10}, z_{11}, \ldots, z_{1s}, \ldots\ \ldots, z_{u0}, z_{u1}, \ldots, z_{us}]$, $l = 1 \ldots, m$, and the coefficients of f_l are given by the initial values X_{0w_1} of the loop variables $X = \{x_1, \ldots, x_m\}$ before $S^{j_1}_{w_1}$.

■

Based on the result of Theorem 6.14, we can now define the *merged closed form system* of the loop variables after an arbitrary sequence $S_{w_1}, \ldots, S_{w_u}$ of u P-solvable inner loops with $u \geq 2$ and $j_1, \ldots, j_u \in \mathbb{N}$ iterations respectively. For simplicity, let us first define for the case of $u = 2$.

Definition 6.15 Let $w_1, w_2 \in \{1, \ldots, k\}$ and $j_1, j_2 \in \mathbb{N}$.

The *merging operation* $\asymp$ over the closed forms of $S^{j_1}_{w_1}$ and $S^{j_2}_{w_2}$ of the loop variables $X = \{x_1, \ldots, x_m\}$ is defined as

$$\mathrm{CF}(S^{j_2}_{w_2},\ E_{S_{w_2}}, X_{w_2}, X_{w_1}) \asymp \mathrm{CF}(S^{j_1}_{w_1},\ E_{S_{w_1}}, X_{w_1}, X_0) =$$

$$\begin{cases} x_1[j_1, j_2] = & f_1(j_1, \theta^{j_1}_{w_1 1}, \ldots, \theta^{j_1}_{w_1 s}, j_2, \theta^{j_2}_{w_2 1}, \ldots, \theta^{j_2}_{w_2 s}) \\ \vdots & \\ x_m[j_1, j_2] = & f_m(j_1, \theta^{j_1}_{w_1 1}, \ldots, \theta^{j_1}_{w_1 s}, j_2, \theta^{j_2}_{w_2 1}, \ldots, \theta^{j_2}_{w_2 s}). \end{cases} \tag{6.29}$$

where

- $x_l[j_1, j_2]$, $l = 1, \ldots, m$, represent the values of the loop variables x_l after respectively j_1 and j_2 iterations of S_{w_1} and S_{w_2}, i.e. after $\underbrace{S_{w_1}; \ldots; S_{w_1}}_{j_1 \text{ times}}; \underbrace{S_{w_2}; \ldots; S_{w_2}}_{j_2 \text{ times}}$;
- X_{w_1} denotes the initial values of the loop variables $X = \{x_1, \ldots, x_m\}$ before $S^{j_2}_{w_2}$ that are the final values of the loop variables after $S^{j_1}_{w_1}$;
- X_{w_2} denotes the final values of the loop variables $X = \{x_1, \ldots, x_m\}$ after $S^{j_2}_{w_2}$, i.e. after $S^{j_1}_{w_1}; S^{j_2}_{w_2}$;
- X_0 are the initial values of the loop variables before $S^{j_1}_{w_1}$, i.e. before $S^{j_1}_{w_1}; S^{j_2}_{w_2}$;
- $E_{S_{w_i}} = \{\theta^{j_i}_{w_i 1}, \ldots, \theta^{j_i}_{w_i s}\}$, $i = 1, 2$;
- $f_l \in \bar{\mathbb{K}}[z_{10}, \ldots, z_{1s}, z_{20}, \ldots, z_{2s}]$, $l = 1, \ldots, m$. The variables $z_{i0}, \ldots, z_{is}$ stand for the polynomially related C-finite sequences $j_i, \theta^{j_i}_{w_i 1}, \ldots, \theta^{j_i}_{w_i s}$, $i = 1, 2$;

- the coefficients of $f_l,\ l = 1,\ldots,m$ are given by the initial values X_0.

We denote by

$$\mathrm{CF}(S^{j_1}_{w_1}; S^{j_2}_{w_2}, E_{S_{w_1};S_{w_2}}, X_2, X_0) := \mathrm{CF}(S^{j_2}_{w_2},\ E_{S_{w_2}}, X_{w_2}, X_{w_1}) \asymp \mathrm{CF}(S^{j_1}_{w_1},\ E_{S_{w_1}}, X_{w_1}, X_0) \tag{6.30}$$

the *merged closed form system* of the closed form systems of $S^{j_1}_{w_1}$ and $S^{j_2}_{w_2}$, where

- X_2 denotes the value of loop variables after the two inner loops, i.e. after $S^{j_1}_{w_1}; S^{j_2}_{w_2}$;
- $E_{S_{w_1};S_{w_2}} = E_{S_{w_1}} \cup E_{S_{w_2}} = \{\theta^{j_1}_{w_1 1}, \ldots, \theta^{j_1}_{w_1 s}, \theta^{j_2}_{w_2 1}, \ldots, \theta^{j_2}_{w_2 s}\}$.

Note that operator $\asymp$ is well-defined (and associative), due to Theorem 6.14. Moreover, Theorem 6.14 gives an algorithmic computation of the merged closed form.

Finally, using Definition 6.15 and Theorem 6.14, we can now inductively define the *merged closed form system* of the loop variables after an arbitrary sequence $S_{w_1}, \ldots, S_{w_u}$ of $u \geq 2$ P-solvable inner loops with $j_1, \ldots, j_u \in \mathbb{N}$ iterations respectively.

Definition 6.16 Let $u \geq 2$, $w_1, \ldots, w_u \in \{1, \ldots, k\}$ and $j_1, \ldots, j_u \in \mathbb{N}$.
The *merging operation* $\asymp$ over the closed forms of $S^{j_1}_{w_1}, \ldots, S^{j_u}_{w_u}$ of the loop variables $X = \{x_1, \ldots, x_m\}$ is defined as

$$\mathrm{CF}(S^{j_u}_{w_u},\ E_{S_{w_u}}, X_{w_u}, X_{w_{u-1}}) \asymp \mathrm{CF}(S^{j_1}_{w_1}; \ldots; S^{j_{u-1}}_{w_{u-1}},\ E_{S_{w_1};\ldots;S_{w_{u-1}}}, X_{w_{u-1}}, X_0) =$$

$$\begin{cases} x_1[j_1, \ldots, j_u] = & f_1(j_1, \theta^{j_1}_{w_1 1}, \ldots, \theta^{j_1}_{w_1 s}, \ldots\ \ldots,\ j_u, \theta^{j_u}_{w_u 1}, \ldots, \theta^{j_u}_{w_u s}) \\ \vdots & \\ x_m[j_1, \ldots, j_u] = & f_m(j_1, \theta^{j_1}_{w_1 1}, \ldots, \theta^{j_1}_{w_1 s}, \ldots\ \ldots,\ j_u, \theta^{j_u}_{w_u 1}, \ldots, \theta^{j_u}_{w_u s}), \end{cases} \tag{6.31}$$

where

- $x_l[j_1, \ldots, j_u]$, $l = 1, \ldots, m$, represent the values of the loop variables x_l after respectively $j_1, \ldots, j_u$ iterations of $S_{w_1}, \ldots, S_{w_u}$, i.e. after $\underbrace{S_{w_1}; \ldots; S_{w_1}}_{j_1 \text{ times}}; \ldots; \underbrace{S_{w_u}; \ldots; S_{w_u}}_{j_u \text{ times}}$;
- $X_{w_{u-1}}$ denotes the initial values of the loop variables $X = \{x_1, \ldots, x_m\}$ before $S^{j_u}_{w_u}$ that are the final values of the loop variables after $S^{j_1}_{w_1}; \ldots; S^{j_{u-1}}_{w_{u-1}}$;
- X_{w_u} denotes the final values of the loop variables $X = \{x_1, \ldots, x_m\}$ after $S^{j_u}_{w_u}$, i.e. after $S^{j_1}_{w_1}; \ldots; S^{j_{u-1}}_{w_{u-1}}; S^{j_u}_{w_u}$;
- X_0 are the initial values of the loop variables before $S^{j_1}_{w_1}$, i.e. before $S^{j_1}_{w_1}; \ldots; S^{j_{u-1}}_{w_{u-1}}; S^{j_u}_{w_u}$;
- $E_{S_{w_i}} = \{\theta^{j_i}_{w_i 1}, \ldots, \theta^{j_i}_{w_i s}\},\ i = 1, \ldots, u$;
- $E_{S_{w_1};\ldots;S_{w_{u-1}}} = E_{S_{w_1}} \cup \cdots \cup E_{S_{w_{u-1}}} = \{\theta^{j_1}_{w_1 1}, \ldots, \theta^{j_1}_{w_1 s}, \ldots\ \ldots, \theta^{j_{u-1}}_{w_{u-1} 1}, \ldots, \theta^{j_{u-1}}_{w_{u-1} s}\}$;

- $f_l \in \bar{\mathbb{K}}[z_{10},\ldots,z_{1s},\ldots,z_{u0},\ldots,z_{us}]$, $l=1,\ldots,m$. The variables $z_{i0},\ldots,z_{is}$ stand for the polynomially related C-finite sequences $j_i, \theta^{j_i}_{w_i1},\ldots,\theta^{j_i}_{w_is}$, $i=1,\ldots,u$;
- the coefficients of f_l are given by the initial values X_0.

We denote by

$$\begin{gathered}\mathrm{CF}(S^{j_1}_{w_1};\ldots;S^{j_{u-1}}_{w_{u-1}};S^{j_u}_{w_u},\ E_{S_{w_1};\ldots;S_{w_{u-1}};S_{w_u}},X_u,X_0) := \\ \mathrm{CF}(S^{j_u}_{w_u},\ E_{S_u},X_{w_u},X_{w_{u-1}}) \asymp \mathrm{CF}(S^{j_1}_{w_1};\ldots;S^{j_{u-1}}_{w_{u-1}},\ E_{S_{w_1};\ldots;S_{w_{u-1}}},X_{w_{u-1}},X_0)\end{gathered} \tag{6.32}$$

the *merged closed form system* of the closed form systems of $S^{j_1}_{w_1},\ldots,S^{j_u}_{w_u}$, where

- X_u denotes the values of loop variables after the sequence of u P-solvable inner loops, i.e. after $S^{j_1}_{w_1};\ldots;S^{j_{u-1}}_{w_{u-1}};S^{j_u}_{w_u}$;
- $E_{S_{w_1};\ldots;S_{w_u}} = E_{S_{w_1};\ldots;S_{w_{u-1}}} \cup E_{S_{w_u}} = \{\theta^{j_1}_{w_11},\ldots,\theta^{j_1}_{w_1s},\ldots\ \ldots,\theta^{j_u}_{w_u1},\ldots,\theta^{j_u}_{w_us}\}$.

Similarly to Definition 6.15, Theorem 6.14 ensures that the operator $\asymp$ is well-defined. Moreover, Theorem 6.14 gives an algorithmic computation of the merged closed form system of an arbitrary sequence of $u \geq 2$ inner loops.

Example 6.17 Using the closed forms of the inner loops S_1 and S_2 presented in Example 6.12, the merged closed form of the sequence of inner loop S_1 and S_2 with j_1 and j_2 iterations respectively is computed based on Definition 6.15, as follows.

For the inner loop sequence $S^{j_1}_1;S^j_2$ the initial values X_{02} are given by the values $X_1=\{a[j_1],b[j_1],d[j_1],y[j_1]\}$ after $S^{j_1}_1$. Hence, by replacing the closed form expressions of $a[j_1],b[j_1],d[j_1],y[j_1]$ in the closed form system of $a[j_2],b[j_2],d[j_2],y[j_2]$, we get the closed form system for the values of loop variables $a[j_1,j_2],b[j_1,j_2],d[j_1,j_2],y[j_1,j_2]$ after $\underbrace{S_1;\ldots;S_1}_{j_1 \text{ times}};\underbrace{S_2;\ldots;S_2}_{j_2 \text{ times}}$.

For simplicity, let us rename the initial values $X_{01}=\{a[0_1],b[0_1],d[0_1],y[0_1]\}$ to respectively $X_0=\{a[0],b[0],d[0],y[0]\}$.

The obtained merged closed form is thus given below.

$$\mathrm{CF}(S^{j_1}_1;S^j_2,E_{S_1;S_2},X,X_0):$$

$$\begin{cases} a[j_1,j_2] &= a[0]+2*b[0]*z_{11}-2b[0]*z_{21}*z_{11} \\ b[j_1,j_2] &= b[0]*z_{22}*z_{11} \\ d[j_1,j_2] &= d[0]*z_{12}*z_{23} \\ y[j_1,j_2] &= y[0]+d[0]*z_{12}-d[0]*z_{24}*z_{12}, \end{cases}$$

with the already computed algebraic dependencies

$$\begin{cases} z_{11} - z_{12} &= 0 \\ z_{21} - z_{24} &= 0 \\ z_{22} - z_{24} &= 0 \\ z_{23} - z_{24} &= 0, \end{cases}$$

where $E_{S_1;S_2} = \{z_{11}, z_{12}, z_{21}, z_{22}, z_{23}, z_{24}\}$ and $X = \{a[j_1, j_2], b[j_1, j_2], d[j_1, j_2], y[j_1, j_2]\}$.

Remark 6.18 The closed forms of the loop variables corresponding to the system of recurrences of a P-solvable inner loop S_{w_i} with the loop counter j_i (see (6.22)) represent solutions of univariate sequences, with the recurrence variable j_i.

Based on Definition 6.16, the merged closed forms of $u \geq 2$ P-solvable inner loops represent solutions of multivariate sequences, with recurrence variables $j_1, \ldots, j_u$.

6.3 Polynomial Relations of Inner Loop Sequences

According to Definition 5.9, as presented also in (6.22), for each P-solvable inner loop S_{w_i} with respectively $j_i \in \mathbb{N}$ iterations, where $w_i \in \{1, \ldots, k\}$ and $i = 1, \ldots, u$, its algebraic exponential sequences in j_i are polynomially related. Moreover, as noted already in (6.22), the ideal of algebraic dependencies of the exponential sequences is determined using Algorithm 4.45. We thus have

$$A_{w_i} = I[j_i, \theta_{w_i 1}^{j_i}, \ldots, \theta_{w_i s}^{j_i}] \trianglelefteq \bar{\mathbb{K}}[z_{i0}, \ldots, z_{is}],$$

where the variables $z_{i0}, \ldots, z_{is}$ stand for the C-finite sequences $j_i, \theta_{w_i 1}^{j_i}, \ldots, \theta_{w_i s}^{j_i}$.

What remains is to determine the ideal A_* of algebraic dependencies of the C-finite sequences $j_1, \theta_{w_1 1}^{j_1}, \ldots, \theta_{w_1 s}^{j_1}, \ldots \ldots, j_u, \theta_{w_u 1}^{j_u}, \ldots, \theta_{w_u s}^{j_u}$ from the merged closed form system (6.25) of $u \geq 2$ P-solvable inner loops, i.e. $A_* = I(j_1, \theta_{w_1 1}^{j_1}, \ldots, \theta_{w_1 s}^{j_1}, \ldots \ldots, j_u, \theta_{w_u 1}^{j_u}, \ldots, \theta_{w_u s}^{j_u})$.

Note that the exponential sequences from the closed form system of a P-solvable inner loop S_{w_i} depend only on the inner loop counter j_i. By Theorem 4.47 we then have

$$A_* = \sum_{i=1}^{u} A_{w_i}. \tag{6.33}$$

Hence, (6.33) ensures that the ideal A_* of algebraic dependencies among the C-finite sequences $j_1, \theta_{w_1 1}^{j_1}, \ldots, \theta_{w_1 s}^{j_1}, \ldots \ldots, j_u, \theta_{w_u 1}^{j_u}, \ldots, \theta_{w_u s}^{j_u}$ from the merged closed form (6.25) is completely determined by the sum of the ideals of algebraic dependencies of $j_i, \theta_{w_i,1}^{j_i}, \ldots, \theta_{w_i,s}^{j_i}$, $i = 1, \ldots, u$. In other words, A_* does not contain any other polynomial relation beyond the ones coming from the ideals A_{w_i}.

We have now all necessary results to state the algorithm for computing the *ideal of all valid polynomial relations among the loop variables X with initial values X_0 after an arbitrary sequence of $u \geq 2$ P-solvable loops $S_{w_1}, \ldots, S_{w_u}$ with assignments only, having respectively $j_1, \ldots, j_u$ iterations, i.e. after $S_{w_1}^{j_1}, \ldots, S_{w_u}^{j_u}$*. More precisely, we can now compute the ideal of all valid polynomial

relations among the loop variables X with initial values X_0 corresponding to the sequence of assignments $\underbrace{S_{w_1};\ldots;S_{w_1}}_{j_1 \text{ times}};\underbrace{S_{w_2};\ldots;S_{w_2}}_{j_2 \text{ times}};\ldots\ \ldots;\underbrace{S_{w_u};\ldots;S_{w_u}}_{j_u \text{ times}}$.

Using Hoare triple notation, we thus compute the ideal of all polynomial relations $p \in \mathbb{K}[X]$ for $S_{w_1}^{j_1},\ldots,S_{w_u}^{j_u}$ among the loop variables X with initial values X_0 such that the following formula involving Hoare triple notation is valid for all $j_1,\ldots,j_u \in \mathbb{N}$:

$$\{p(X) = 0 \wedge X = X_0\} \quad S_{w_1}^{j_1};\ldots;S_{w_u}^{j_u} \quad \{p(X) = 0\}. \tag{6.34}$$

In other words, we are trying to achieve

$$\{p(X) = 0 \wedge X = X_0\} \quad S_1^* * \cdots * S_u^* \quad \{p(X) = 0\}.$$

Algorithm 6.19 (Polynomial Relations for Sequence of P-solvable Loops with Assignments)
Input: u P-solvable (inner) loops $S_{w_1},\ldots,S_{w_u}$ with $j_1,\ldots,j_u$ iterations
Output: The ideal $G \trianglelefteq \mathbb{K}[X]$ of polynomial relations among the loop variables X with initial values X_0 after $S_{w_1}^{j_1};\ldots;S_{w_u}^{j_u}$
Assumption: S_{w_i} are sequences of assignments, $u \geq 1,\ w_1,\ldots,w_u \in \{1,\ldots,k\},\ j_1,\ldots,j_u \in \mathbb{N}$

1 **for** each $S_{w_i}^{j_i},\ i = 1,\ldots,u$ **do**
2 Apply steps 1, 2 and 3 of Algorithm 5.11 for determining $\mathrm{CF}(S_{w_i}^{j_i},\ E_{S_{w_i}},X_i,X_{0i})$
3 Apply Algorithm 4.45 to get $A_{w_i} = I(j_i,\theta_{w_i1}^{j_i},\ldots,\theta_{w_is}^{j_i})$
4 **endfor**
5 By Definition 6.16 and (6.32), compute the merged closed form of $S_{w_1}^{j_1};\ldots;S_{w_u}^{j_u}$:

$$\mathrm{CF}(S_{w_1}^{j_1};\ldots;S_{w_u}^{j_k},\ E_{S_{w_1};\ldots;S_{w_u}},X_u,X_0):$$

$$\begin{cases} x_1[j_1,\ldots,j_u] = & f_1(j_1,\theta_{w_11}^{j_1},\ldots,\theta_{w_1s}^{j_1},\ldots\ \ldots,j_u,\theta_{w_u1}^{j_u},\ldots,\theta_{w_us}^{j_u}) \\ \vdots & \\ x_m[j_1,\ldots,j_u] = & f_m(j_1,\theta_{w_11}^{j_1},\ldots,\theta_{w_1s}^{j_1},\ldots\ \ldots,j_u,\theta_{w_u1}^{j_u},\ldots,\theta_{w_us}^{j_u}), \end{cases}$$

where $f_l \in \bar{\mathbb{K}}[z_{10},\ldots,z_{1s},\ldots\ \ldots,z_{u0},\ldots,z_{us}]$;
the variables $z_{i0},\ldots,z_{is}$ are standing for the C-finite sequences $j_i,\theta_{w_i1}^{j_i},\ldots,\theta_{w_is}^{j_i},\ i = 1,\ldots,u$;
the coefficients of f_l are given by the initial values X_0 (that are the initial values X_{01})

6 $A_* = \sum_{i=1}^{u} A_{w_i}$
7 $I = \langle x_1 - f_1,\ldots,x_m - f_m\rangle + A_* \subset \bar{\mathbb{K}}[z_{10},\ldots,z_{1s},\ldots,\ldots,z_{u0},\ldots,z_{us},x_1,\ldots,x_m]$
8 **return** $G = I \cap \mathbb{K}[x_1,\ldots,x_m]$.

Similarly to Algorithm 4.14, elimination of the variables $z_{10},\ldots,z_{us}$ at step 8 is performed by applying Algorithm 4.8 to compute the Gröbner basis of I w.r.t. an elimination order $\succ$ such that

$$z_{10} \succ \cdots \succ z_{1s} \succ \cdots\ \cdots \succ z_{u0} \succ \cdots \succ z_{us} \succ x_1 \cdots \succ x_m.$$

Hence, Algorithm 6.19 computes algorithmically the set of generators for the ideal G.

In step 5 of Algorithm 6.19, in the case $u = 1$, i.e. when there is a single P-solvable loop, no merging is performed. Moreover, if $u = 1$ then Algorithm 6.19 becomes Algorithm 5.11, and thus Algorithm 6.19 is a generalization of Algorithm 5.11.

Theorem 6.20 Algorithm 6.19 is correct. That is, it finds the generators of the ideal G of polynomial relations among the values of loop variables $X = \{x_1, \ldots, x_m\}$ with initial values X_0 after an arbitrary sequence of $u \geq 1$ P-solvable inner loops.

Proof. The proof is similar to the correctness proof of Algorithm 5.11, presented in the proof Theorem 5.13.

By correctness of Algorithms 5.11 and 4.45, we have that $A_{w_i},\ i = 1, \ldots, u$, is the ideal of the algebraic dependencies among $j_i, \theta^{j_i}_{w_i 1}, \ldots, \theta^{j_i}_{w_i s}$ (step 3 of Algorithm 6.19).

The C-finite sequences $j_i, \theta^{j_i}_{w_i 1}, \ldots, \theta^{j_i}_{w_i s}$ depend only on j_i. By Theorem 4.47 and (6.33), this implies that $A_* = \sum_{i=1}^{u} A_{w_i}$ is the ideal of *all* algebraic dependencies among all C-finite sequences from the merged closed form system, i.e. among $j_1, \theta^{j_1}_{w_1 1}, \ldots, \theta^{j_1}_{w_1 s}, \ldots\ \ldots, j_u, \theta^{j_u}_{w_u 1}, \ldots, \theta^{j_u}_{w_u s}$ (step 6 of Algorithm 6.19).

Based on the merged closed form computed at step 5, let us consider the polynomial map below.

$$h : \bar{\mathbb{K}}[x_1, \ldots, x_m] \to \bar{\mathbb{K}}[z_{10}, \ldots, z_{us}]/A_*,$$

$$h(x_l) = f_l + A_*, \quad l = 1, \ldots, m,$$

$$h(c) = c + A_* \quad \text{for } c \in \bar{\mathbb{K}},$$

where the coefficients of $f_l \in \bar{\mathbb{K}}[z_{10}, \ldots, z_{us}]$ are determined by the initial values X_0.

From the merged closed form representation of $x_1, \ldots, x_m$ computed at step 5 of Algorithm 6.19, the kernel of the polynomial map h is the ideal of all polynomial relations among the values of loop variables $X = \{x_1, \ldots, x_m\}$ after the considered sequence of $u \geq 1$ P-solvable inner loops with initial values X_0.

By results of (Adams and Loustaunau, 1994) and Theorem 4.13, the kernel of h is thus the ideal G computed at steps 6-8 of Algorithm 6.19. We then have

$$G = \ker h = I(x_1, \ldots, x_m) \trianglelefteq \bar{\mathbb{K}}[x_1, \ldots, x_m].$$

Further, note that although the closed form solutions of $x_l,\ l = 1, \ldots, m$, lie in the algebraic extension field $\bar{\mathbb{K}}$, the recurrences of x_l corresponding to the assignment statements of the loop are defined over $\mathbb{K}$. Based on (Kauers and Zimmermann (2006), Lemma 2 and Theorem 1), we thus derive

$$G \trianglelefteq \mathbb{K}[x_1, \ldots, x_m].$$

In other words, although the closed form solutions of x_l lie in the algebraic extension $\bar{\mathbb{K}}[z_{10}, \ldots, z_{us}]$, the (generators of the) ideal G of polynomial relations among the loop variables $X = \{x_1, \ldots, x_m\}$ with initial values X_0 after $S^{j_1}_{w_1}; \ldots; S^{j_u}_{w_u}$ is an ideal of $\mathbb{K}[x_1, \ldots, x_m]$.

■

Example 6.21

In Example 6.17, the merged closed form of $S_1^{j_1};S_2^{j_2}$ was already computed. Moreover, we also have a complete characterization of the ideal of algebraic dependencies among $z_{11}, z_{12}, z_{21}, z_{22}, z_{23}$, z_{24}, namely:

$$A_* = \langle z_{11} - z_{12},\ z_{21} - z_{24},\ z_{22} - z_{24},\ z_{23} - z_{24} \rangle$$

According to step 7 of Algorithm 6.19, let us denote $a[j_1, j_2], b[j_1, j_2], d[j_1, j_2], y[j_1, j_2]$ by a, b, d, y. We then have

$$\begin{aligned} I = \quad & \langle\ a - a[0] - 2 * b[0] * z_{11} + 2b[0] * z_{21} * z_{11}, \\ & b - b[0] * z_{22} * z_{11}, \\ & d - d[0] * z_{12} * z_{23}, \\ & y - y[0] - d[0] * z_{12} + d[0] * z_{24} * z_{12}, \\ & z_{11} - z_{12}, \\ & z_{21} - z_{24}, \\ & z_{22} - z_{24}, \\ & z_{23} - z_{24} \rangle. \end{aligned}$$

Finally, by eliminating the variables $z_{11}, z_{12}, z_{21}, z_{22}, z_{23}, z_{24}$ from I by Gröbner basis computation, the obtained ideal of polynomial relations among the loop variables a, b, d, y with initial values $a[0], b[0], d[0], y[0]$ corresponding to $S_1^{j_1};S_2^{j_2}$ is

$$\begin{aligned} G = \quad & \langle\ -b[0] * d + b * d[0], \\ & a * d[0] - a[0] * d[0] - 2 * b[0] * y + 2 * b[0] * y[0], \\ & a * d - a[0] * d - 2 * b * y + 2 * b * y[0] \rangle. \end{aligned}$$

As discussed in Remark 6.13, in the general case of having a P-solvable loop with $k \geq 1$ conditional branches (all tests being ignored), by applying Theorem 6.6 we obtain an outer loop with k P-solvable inner loops $S_1, \ldots, S_k$. Thus an arbitrary iteration of the outer loop is described by an arbitrary sequence (from IS_k) of the k P-solvable inner loops.

By applying Algorithm 6.19 for such an arbitrary sequence of k P-solvable inner loop (i.e. $u = k$ and $(w_1, \ldots, w_k) \in \mathfrak{S}_k$), we obtain the ideal of polynomial relations among the loop variables X with initial values X_0 after an arbitrary sequence of k inner loops.

In order to get all polynomial relations among the loop variables X with initial values X_0 corresponding to an arbitrary iteration of the outer loop, we need to apply Algorithm 6.19 on each possible sequence of k inner loops that are in a number of $k!$ (see Remark 6.13). This way, by correctness of Algorithm 6.19, for each sequence of k inner loops we get the ideal of all their polynomial relations among the loop variables X with initial values X_0 (step 3 of Algorithm 6.22). Using ideal theoretic results (see Theorem 4.10), by taking the *intersection* of all these ideals, we obtain *the ideal of all valid polynomial relations among the loop variables X with initial values X_0 that are valid after any sequence of k P-solvable inner loops* (step 4 of Algorithm 6.22). Based on Remark 6.13, the intersection ideal thus obtained is the ideal of polynomial relations among

the loop variables X with initial values X_0 *after an arbitrary iteration of the outer loop*. Hence, using (6.34), we obtain all valid polynomial relations $p \in \mathbb{K}[X]$ among the loop variables X with initial values X_0 for which the following formula involving Hoare triple notation holds:

$$\{p(X) = 0 \wedge X = X_0\}\ S_1^* * \cdots * S_k^*\ \{p(X) = 0\}. \tag{6.35}$$

This can be algorithmically computed as follows.

Algorithm 6.22 (Polynomial Relations for an Iteration of a Loop having $k \geq 1$ P-solvable Loops with Assignments)
Input: Imperative loop having only $k \geq 1$ P-solvable inner loops $S_1, \ldots, S_k$ with respectively $j_1, \ldots, j_k \in \mathbb{N}$ iterations
Output: The ideal $PI \subset \mathbb{K}[X]$ of the polynomial relations among the loop variables X with initial values X_0 corresponding to an arbitrary iteration of the outer loop
Assumption: X_0 are the initial values of the loop variables X before the arbitrary iteration of the outer loop, and $S_i, i = 1, \ldots, k$, are sequences of assignments

1 $PI =$ Algorithm 6.19$\big(S_1^{j_1},\ \ldots,\ S_k^{j_k}\big)$
2 **for** each $W \in \mathfrak{S}_k \setminus \{(1, \ldots, k)\}$ **do**
3 $\quad G =$ Algorithm 6.19$\big(S_{w_1}^{j_{w_1}},\ \ldots,\ S_{w_k}^{j_{w_k}}\big)$
4 $\quad PI = PI \cap G$
5 **endfor**
6 **return** PI.

Theorem 6.23 Algorithm 6.22 is correct. That is, it returns the generators for the ideal PI of all polynomial relations among the loop variables X with initial values X_0 corresponding to an arbitrary iteration of a loop having only $k \geq 1$ P-solvable loops with assignments.

In other words, PI contains all polynomial relations among the loop variables X with initial values X_0 that are valid after any sequence of k P-solvable inner loops describing a possible iteration of the outer loop.

Proof. Clear by previous discussion and correctness of Algorithm 6.19. ∎

Remark 6.24 Note that for a P-solvable loop with assignments only, i.e. $k = 1$, only step 1 of Algorithm 6.22 is executed, using Algorithm 6.19. Moreover, as already mentioned, for $k = 1$ Algorithm 6.19 becomes Algorithm 5.11, and thus Algorithm 6.22 generalizes Algorithm 5.11.

Example 6.25

From Example 6.21 we already have the ideal of polynomial relations among the loop variables a, b, d, y with initial values $a[0], b[0], d[0], y[0]$ corresponding to $S_1^{j_1}; S_2^{j_2}$, it is computed at step 1 of Algorithm 6.22.

Using the closed forms of the inner loops S_1 and S_2 from Example 6.12, the ideal of polynomial relations among the loop variables a,b,d,y with initial values $a[0],b[0],d[0],y[0]$ corresponding to $S_2^{j_2};S_1^{j_1}$ is determined similarly to Example 6.21, and we derive

$$\begin{aligned} G = \quad & \langle\, -b[0]*d+b*d[0], \\ & a*d[0]-a[0]*d[0]-2*b[0]*y+2*b[0]*y[0], \\ & a*d-a[0]*d-2*b*y+2*b*y[0]\rangle. \end{aligned}$$

Taking the intersection of the ideals of polynomial relations corresponding respectively to $S_1^{j_1};S_2^{j_2}$ and $S_2^{j_2};S_1^{j_1}$, we obtain the ideal of polynomial relations the loop variables a,b,d,y with initial values $a[0],b[0],d[0],y[0]$ corresponding to an arbitrary iteration of the outer loop. We thus have

$$\begin{aligned} PI = \quad & \langle\, -b[0]*d+b*d[0], \\ & a*d[0]-a[0]*d[0]-2*b[0]*y+2*b[0]*y[0], \\ & a*d-a[0]*d-2*b*y+2*b*y[0]\rangle. \end{aligned}$$

What remains is to identify the relationship between the polynomial invariants among the loop variables X of the *outer loop* and the computed polynomial relations using Algorithm 6.22 for *an arbitrary iteration of the outer loop* with only $k \geq 1$ P-solvable inner loops.
The rest of this section addresses this issue.

In what follows, before presenting the remaining steps of our invariant generation algorithm, let us first underline the significance of the merging operation on closed forms (performed by Algorithm 6.19) for deriving valid polynomial relations for a sequence of P-solvable inner loops.
For simplicity, let us fix some notation.

- $I_i \trianglelefteq \mathbb{K}[x_1,\ldots,x_m]$, where $i \in \{1,\ldots,u\}$, $w_1,\ldots,w_u \in \{1,\ldots,k\}$ and $u \geq 1$, denotes the ideal of polynomial relations among the loop variables $X = \{x_1,\ldots,x_m\}$ and initial values $X_0 = \{x_{01},\ldots,x_{0m}\}$ for the inner loop S_{w_i} with $j_i \in \mathbb{N}$ iterations. Hence, I_i is the ideal of polynomial relations among $x_1,\ldots,x_m$ after $S_{w_i}^{j_i}$, parameterized by the initial values X_0.
- $I_i' \trianglelefteq \mathbb{K}[y_{i1},\ldots,y_{im}]$, where $i \in \{1,\ldots,u\}$, $w_1,\ldots,w_u \in \{1,\ldots,k\}$ and $u \geq 1$, denotes the ideal I_i where the loop variables $x_1,\ldots,x_m$ are renamed to $y_{i1},\ldots,y_{im}$. Hence, I_i' is the ideal of polynomial relations among the loop variables $y_{i1},\ldots,y_{im}$ and their initial values $X_0 = \{x_{01},\ldots,x_{0m}\}$ for the inner loop S_{w_i} with $j_i \in \mathbb{N}$ iterations, where y_{il}, $l = 1,\ldots,m$, stand for the variables x_l after $S_{w_i}^{j_i}$. I_i' is thus parameterized by X_0, and we explicitly write $I_i'(y_{i1},\ldots,y_{im},X_0)$.
- $I_i'' \trianglelefteq \mathbb{K}[y_{i1},\ldots,y_{im}]$, where $i \in \{2,\ldots,u\}$, $w_1,\ldots,w_u \in \{1,\ldots,k\}$ and $u \geq 1$, denotes the ideal I_i' where the initial values $x_{01},\ldots,x_{0m}$ are renamed to $y_{i-1\ 1},\ldots,y_{i-1\ m}$. Hence, I_i'' is the ideal of polynomial relations among the loop variables $y_{i1},\ldots,y_{im}$ for the inner loop S_{w_i} with $j_i \in \mathbb{N}$ iterations, and with initial values $y_{i-1\ 1},\ldots,y_{i-1\ m}$. I_i'' is thus parameterized by $y_{i-1\ 1},\ldots,y_{i-1\ m}$, and we explicitly write $I_i''(y_{i1},\ldots,y_{im},y_{i-1\ 1},\ldots,y_{i-1\ m})$.

The relation between the polynomial ideal obtained for a sequence of P-solvable inner loops and the polynomial ideals of each loop from the loop sequence is presented in the theorem below.

Theorem 6.26

Let $u = 2$. W.l.o.g., we consider $w_1 = 1,\ w_2 = 2$ and the two P-solvable inner loops S_{w_1} and S_{w_2}, i.e. S_1 and S_2, with respectively $j_1,\ j_2 \in \mathbb{N}$ iterations.

Let I_*= Algorithm 6.19$(S_1^{j_1},\ S_2^{j_2})$ be the ideal of all polynomial relations among the loop variables $x_1,\ldots,x_m$ after $S_1^{j_1};S_2^{j_2}$, with initial values $X_0 = \{x_{01},\ldots,x_{0m}\}$. Thus I_* is parameterized by X_0.

Let us denote by $\tilde{I}_2$ the ideal I_2'' corresponding to $S_2^{j_2}$, where $y_{21},\ldots,y_{2m}$ are replaced by $x_1,\ldots,x_m$. Then

$$I_1'(y_{11},\ldots,y_{1m},X_0) + \tilde{I}_2(x_1,\ldots,x_m,y_{11},\ldots,y_{1m}) \cap \mathbb{K}[x_1,\ldots,x_m] \subseteq I_*. \tag{6.36}$$

Proof. For proving (6.36),

(1) first we derive a basis for each of the ideals I_1', $\tilde{I}_2$ and I_*;

(2) then we prove (6.36) by using ideal theoretic operations over these bases.

Using (6.23), the closed form system of $S_1^{j_1}$ is

$$\mathrm{CF}(S_1^{j_1},\ E_{S_1},X_1,X_{01}):\quad \begin{cases} x_1[j_1] = & q_{1,1}(j_1,\theta_{11}^{j_1},\ldots,\theta_{1s}^{j_1}) \\ \vdots & \\ x_m[j_1] = & q_{1,m}(j_1,\theta_{11}^{j_1},\ldots,\theta_{1s}^{j_1}), \end{cases} \tag{6.37}$$

where $q_{1,l} \in \bar{\mathbb{K}}[z_{10},z_{11},\ldots,z_{1s}],\ l = 1,\ldots,m$, and the coefficients of $q_{1,l}$ are given by the initial values X_{01}. The variables $z_{10},z_{11},\ldots,z_{1s}$ stand for the polynomially related C-finite sequences $j_1,\theta_{11}^{j_1},\ldots,\theta_{1s}^{j_1}$, and let $A_1 = I(j_1,\theta_{11}^{j_1},\ldots,\theta_{1s}^{j_1}) = I(z_{10},z_{11},\ldots,z_{1s})$ be the ideal of algebraic dependencies among $z_{10},z_{11},\ldots,z_{1s}$.

According to the already introduced notation for I_1', we denote $y_{11} = x_1[j_1],\ldots,y_{1m} = x_m[j_1]$, whereas the initial values X_{01} are denoted by X_0. Hence, (6.37) can be written equivalently as

$$\mathrm{CF}(S_1^{j_1},\ E_{S_1},Y_1,X_0):\quad \begin{cases} y_{11} = & q_{1,1}(z_{10},z_{11},\ldots,z_{1s}) \\ \vdots & \\ y_{1m} = & q_{1,m}(z_{10},z_{11},\ldots,z_{1s}), \end{cases} \tag{6.38}$$

where $Y_1 = \{y_{11},\ldots,y_{1m}\}$.

By correctness of Algorithm 5.11, the ideal I_1' of all polynomial relations among $y_{11},\ldots,y_{1m}$ with initial values X_0 is computed from (6.38) and A_1, by eliminating $z_{10},\ldots,z_{1s}$. We thus have

$$I_1'(Y_1,X_0) = \langle y_{11} - q_{1,1},\ \ldots,\ y_{1m} - q_{1,m}\rangle + A_1 \cap \mathbb{K}[Y_1,X_0].$$

Similarly, using (6.23), the closed form system of $S_2^{j_2}$ is

$$\mathrm{CF}(S_2^{j_2},\ E_{S_2},X_2,X_{02}):\quad \begin{cases} x_1[j_2] = & q_{2,1}(j_2,\theta_{21}^{j_2},\ldots,\theta_{2s}^{j_2}) \\ \vdots & \\ x_m[j_2] = & q_{2,m}(j_2,\theta_{21}^{j_2},\ldots,\theta_{2s}^{j_2}), \end{cases} \tag{6.39}$$

where $q_{2,l} \in \bar{\mathbb{K}}[z_{20}, z_{21}, \ldots, z_{2s}]$, $l = 1, \ldots, m$, and the coefficients of $q_{2,l}$ are given by the initial values X_{02}. The variables $z_{20}, z_{21}, \ldots, z_{2s}$ stand for the polynomially related C-finite sequences $j_2, \theta_{21}^{j_2}, \ldots, \theta_{2s}^{j_2}$, and let $A_2 = I(j_2, \theta_{21}^{j_2}, \ldots, \theta_{2s}^{j_2}) = I(z_{20}, z_{21}, \ldots, z_{2s})$ be the ideal of algebraic dependencies among $z_{20}, z_{21}, \ldots, z_{2s}$.

According to the already introduced notation for I_2', we denote $y_{21} = x_1[j_2], \ldots, y_{2m} = x_m[j_2]$, whereas the initial values X_{02} are denoted by X_0. Hence, (6.39) can be written equivalently as

$$\mathrm{CF}(S_2^{j_2},\, E_{S_2}, Y_2, X_0): \quad \begin{cases} y_{21} = & q_{2,1}(z_{20}, z_{21}, \ldots, z_{2s}) \\ \vdots & \\ y_{2m} = & q_{2,m}(z_{20}, z_{21}, \ldots, z_{2s}), \end{cases} \tag{6.40}$$

where $Y_2 = \{y_{21}, \ldots, y_{2m}\}$.

By correctness of Algorithm 5.11, the ideal I_2' of all polynomial relations among $y_{21}, \ldots, y_{2m}$ with initial values X_0 is computed from (6.40) and A_2, by eliminating $z_{20}, \ldots, z_{2s}$. We then have

$$I_2'(Y_2, X_0) = \langle y_{21} - q_{2,1},\ \ldots,\ y_{2m} - q_{2,m} \rangle + A_2 \cap \mathbb{K}[Y_2, X_0].$$

The ideal I_2'' is the ideal of polynomial relations among $y_{21}, \ldots, y_{2m}$ with initial values $y_{11}, \ldots, y_{1m}$ obtained from I_2' by renaming the initial values X_0 to Y_1. We thus derive

$$I_2''(Y_2, Y_1) = \langle y_{21} - q_{2,1},\ \ldots,\ y_{2m} - q_{2,m} \rangle + A_2 \cap \mathbb{K}[Y_2, Y_1].$$

The ideal $\tilde{I}_2$ of polynomial relations among $x_1, \ldots, x_m$ with initial values by $y_{11}, \ldots, y_{1m}$ is obtained from I_2'' by renaming the variables $y_{21}, \ldots, y_{2m}$ to $x_1, \ldots, x_m$. We then finally obtain

$$\tilde{I}_2(X, Y_1) = \langle x_1 - q_{2,1},\ \ldots,\ x_m - q_{2,m} \rangle + A_2 \cap \mathbb{K}[X, Y_1].$$

By Theorem 6.14 and Definition 6.15, the merged closed form of $S_1^{j_1}; S_2^{j_2}$ is computed based on the fact that the initial values of $S_2^{j_2}$ are given by the final values of $S_1^{j_1}$. Note, that I_2'' (or $\tilde{I}_2$) reflects this fact since it is parameterized by the initial values Y_1 that are the final values after $S_1^{j_1}$. Moreover, by (6.25) and (6.29), the merged closed form system of $x_1, \ldots, x_m$ after $S_1^{j_1}; S_2^{j_2}$ is

$$\mathrm{CF}(S_1^{j_1}; S_2^{j_2},\, E_{S_1;S_2}, X, X_0) =$$

$$\begin{cases} x_1 = & f_1(z_{10}, \ldots z_{1s}, z_{20}, \ldots, z_{2s}) \\ \vdots & \\ x_m = & f_m(z_{10}, \ldots z_{1s}, z_{20}, \ldots, z_{2s}), \end{cases} \tag{6.41}$$

where $f_l \in \bar{\mathbb{K}}[z_{10}, \ldots, z_{1s}, z_{20}, \ldots, z_{2s}]$, $l = 1, \ldots, m$, and the coefficients of f_l are given by the initial values X_0. The variables $z_{i0}, z_{i1}, \ldots, z_{is}$ stand for the polynomially related C-finite sequences $j_i, \theta_{i1}^{j_i}, \ldots, \theta_{is}^{j_i}, i = 1, 2$.

Further, by Theorem 4.47 and Remark 6.18, $A_* = I(z_{10},\ldots,z_{1s},z_{20},\ldots,z_{2s}) = A_1 + A_2$ is the ideal of all algebraic dependencies among $z_{10},\ldots,z_{1s},z_{20},\ldots,z_{2s}$.

By correctness of Algorithm 6.19, the ideal I_* of all polynomial relations among the loop variables $x_1,\ldots,x_m$ with initial values X_0 after $S_1^{j_1};S_2^{j_2}$ is then computed from (6.41) and A_*, by eliminating $z_{10},\ldots,z_{1s},z_{20},\ldots,z_{2s}$. Namely, we have

$$I_*(X,X_0) = \langle x_1 - f_1,\ \ldots,\ x_m - f_m\rangle + A_* \cap \mathbb{K}[X,X_0].$$

In what follows, we show

$$(I_1' + \tilde{I}_2) \cap \mathbb{K}[X] \subseteq I_*.$$

For simplicity, we denote

$$\begin{aligned} B_1 = &\quad \langle y_{11} - q_{1,1},\ \ldots,\ y_{1m} - q_{1,m}\rangle + A_1,\\ B_2 = &\quad \langle x_1 - q_{2,1},\ \ldots,\ x_m - q_{2,m}\rangle + A_2. \end{aligned}$$

Note that I_1' is parameterized only by X_0 and is free of X, whereas $\tilde{I}_2$ is parameterized only by Y_1 and it is free of X_0. W.l.o.g., we can thus use the same set of variables for computing the elimination ideals I_1' and $\tilde{I}_2$, and we have

$$\begin{aligned} I_1'(Y_1,X_0) &= B_1 \cap \mathbb{K}[X,Y_1,X_0],\\ \tilde{I}_2(X,Y_1) &= B_2 \cap \mathbb{K}[X,Y_1,X_0]. \end{aligned}$$

Thus

$$\begin{aligned} I_1'(Y_1,X_0) + \tilde{I}_2(X,Y_1) = &\quad (B_1 \cap \mathbb{K}[X,Y_1,X_0]) + (B_2 \cap \mathbb{K}[X,Y_1,X_0])\\ \subseteq &\quad (B_1 + B_2) \cap \mathbb{K}[X,Y_1,X_0]. \end{aligned} \tag{6.42}$$

Further, by Theorem 6.14 and (6.41), we derive

$$\begin{aligned} &\langle x_1 - q_{2,1}(z_{20},\ldots,z_{2s},Y_1),\ldots,x_m - q_{2,m}(z_{20},\ldots,z_{2s},Y_1),\\ &y_{11} - q_{1,1}(z_{10},\ldots,z_{1s},X_0),\ldots,y_{1m} - q_{1,m}(z_{10},\ldots,z_{1s},X_0)\rangle + A_2 + A_1 \cap\ \mathbb{K}[X,X_0] =\\ &B_1 + B_2 \cap \mathbb{K}[X,X_0] =\\ &\langle x_1 - q_{2,1}\big(z_{20},\ldots,z_{2s},q_{1,1}(z_{10},\ldots,z_{1s},X_0)\big),\ldots,\\ &x_m - q_{2,m}\big(z_{20},\ldots,z_{2s},q_{1,1}(z_{10},\ldots,z_{1s},X_0)\big),\\ &y_{11} - q_{1,1}(z_{10},\ldots,z_{1s},X_0),\ldots,y_{1m} - q_{1,m}(z_{10},\ldots,z_{1s},X_0)\rangle + A_* \cap\ \mathbb{K}[X,X_0] =\\ &\langle x_1 - q_{2,1}\big(z_{20},\ldots,z_{2s},q_{1,1}(z_{10},\ldots,z_{1s},X_0)\big),\ldots,\\ &x_m - q_{2,m}\big(z_{20},\ldots,z_{2s},q_{1,1}(z_{10},\ldots,z_{1s},X_0)\big)\rangle + A_* \cap\ \mathbb{K}[X,X_0] =\\ &\langle x_1 - f_1(z_{10},\ldots,z1s,z_{20},\ldots,z_{2s},X_0),\ldots,\\ &x_m - f_m(z_{10},\ldots,z1s,z_{20},\ldots,z_{2s},X_0)\rangle + A_* \cap\ \mathbb{K}[X,X_0] =\\ &I_*. \end{aligned} \tag{6.43}$$

By (6.43) and (6.42), we finally have

$$\begin{aligned} I_1' + \tilde{I}_2 \cap \mathbb{K}[X, X_0] \subseteq & \quad ((B_1 + B_2) \cap \mathbb{K}[X, Y_1, X_0]) \cap \mathbb{K}[X, X0] \\ & = (B_1 + B_2) \cap \mathbb{K}[X, X0] \\ & = I_*, \end{aligned}$$

and thus

$$I_1' + \tilde{I}_2 \cap \mathbb{K}[X] \subseteq I_*.$$

■

Corollary 6.27

Let $u = 2$. W.l.o.g., we consider $w_1 = 2,\ w_2 = 1$, and the two P-solvable inner loops S_{w_1} and S_{w_2}, i.e. S_2 and S_1, with respectively $j_1,\ j_2 \in \mathbb{N}$ iterations.

Let I_*= Algorithm 6.19$(S_2^{j_1},\ S_1^{j_2})$ be the ideal of all polynomial relations among the loop variables $x_1, \ldots, x_m$ after $S_2^{j_1}; S_1^{j_2}$, with initial values $X_0 = \{x_{01}, \ldots, x_{0m}\}$. Thus I_* is parameterized by X_0.

Let us denote by $\tilde{I}_2$ the ideal I_2'' corresponding to $S_1^{j_2}$, where $y_{21}, \ldots, y_{2m}$ are replaced by $x_1, \ldots, x_m$.

Then

$$I_1'(y_{11}, \ldots, y_{1m}, X_0) + \tilde{I}_2(X, y_{11}, \ldots, y_{1m}) \cap \mathbb{K}[X] \subseteq I_*. \tag{6.44}$$

Proof. It obviously follows from the well-definement of the merging operator $\asymp$ and by interchanging the orders of S_1 and S_2 in Theorem 6.26.

■

The inclusion in (6.36) is proper, as it is shown by the example below.

Example 6.28 Consider the imperative loop

```
        While[...,
S1 :        If[...Then b := b/2; d := d/2]
            []
S2 :        If[...Then b := b/2; d := d/2]].
```

By applying Algorithm 6.19 for j_1 iterations of S_1 and j_2 iterations of S_2, i.e. Algorithm 6.19$(S_1^{j_1}, S_2^{j_2})$, we get

$$I_* = \langle \{b * d[0] - d * b[0]\} \rangle,$$

where $b[0],\ d[0]$ are the initial values of $b,\ d$.

Further, by applying Algorithm 5.11 (or Algorithm 6.19) separately on $S_1^{j_1}$ and $S_2^{j_2}$, we get

$$I_1'(b_1, d_1, b[0], d[0]) = \langle \{b_1 * d[0] - d_1 * b[0]\} \rangle$$

and

$$I_2''(b_2, d_2, b_1, d_1) = \langle \{b_2 * d_1 - d_2 * b_1\} \rangle.$$

Thus

$$\tilde{I}_2(b, d, b_1, d_1) = \langle \{b * d_1 - d * b_1\} \rangle,$$

yielding that

$$I_1'(b_1,d_1,b[0],d[0]) + \tilde{I}_2(b,d,b_1,d_1) \cap \mathbb{K}[b,d] = \emptyset.$$

The result of Theorem 6.26 can be generalized to an arbitrary sequence $S_{w_1},\ldots,S_{w_u}$ of u P-solvable inner loops with $j_1,\ldots,j_u$ iterations respectively, as follows.

Theorem 6.29

Consider $u \geq 1$, $w_1,\ldots,w_u \in \{1,\ldots,k\}$, $j_1,\ldots,j_u \in \mathbb{N}$.

Let I_*= Algorithm 6.19($S_{w_1}^{j_1},\ldots,S_{w_u}^{j_u}$) be the ideal of all polynomial relations among the loop variables $x_1,\ldots,x_m$ after an arbitrary sequence $S_{w_1},\ldots,S_{w_u}$ of u P-solvable inner loops with $j_1,\ldots,j_u$ iterations respectively, i.e. after $S_{w_1}^{j_1};\ldots;S_{w_u}^{j_u}$, with initial values $X_0 = \{x_{01},\ldots,x_{0m}\}$.

Let us denote by $\tilde{I}_u$ the ideal I_u'' corresponding to $S_{w_u}^{j_u}$ where $y_{u1},\ldots,y_{um}$ are replaced by $x_1,\ldots,x_m$.

Then

$$\begin{aligned} &I_1'(y_{11},\ldots,y_{1m},X_0)+\\ &I_2''(y_{21},\ldots,y_{2m},y_{11},\ldots,y_{1m})+\\ &\cdots\ \cdots+ \qquad\qquad \subseteq I_*. \\ &I_{u-1}''(y_{u-1\,1},\ldots,y_{u-1\,m},y_{u-2\,1},\ldots,y_{u-2\,m})+\\ &\tilde{I}_u(X,y_{u-1\,1},\ldots,y_{u-1\,m}) \cap \mathbb{K}[X] \end{aligned} \tag{6.45}$$

Proof. We prove this by induction over u.

Induction base: $u = 1$. There is nothing to prove.

Induction step.

By Theorem 6.14 and Definition 6.16 (note that $\asymp$ is associative), the closed forms of the loop variables after $S_{w_1}^{j_1};\ldots;S_{w_{u-1}}^{j_{u-1}}$ are polynomials in the loop counters $j_1,\ldots,j_{u-1}$ and algebraically related exponential sequences in $j_1,\ldots,j_{u-1}$.

W.l.o.g., we keep the notation for the variables $x_1,\ldots,x_m$ after $S_{w_1}^{j_1};\ldots;S_{w_{u-1}}^{j_{u-1}}$ as $y_{u-1\,1},\ldots,y_{u-1\,m}$. Let $\tilde{I}_*$ be the ideal of polynomial relations among $x_1,\ldots,x_m$, i.e. among $y_{u-1\,1},\ldots,y_{u-1\,m}$ after $S_{w_1}^{j_1};\ldots;S_{w_{u-1}}^{j_{u-1}}$, with initial values X_0. Thus, by the induction hypothesis we have

$$\begin{aligned} &I_1'(y_{11},\ldots,y_{1m},X_0)+\\ &I_2''(y_{21},\ldots,y_{2m},y_{11},\ldots,y_{1m})+\\ &\cdots\ \cdots+ \qquad\qquad \subseteq \tilde{I}_*(y_{u-1\,1},\ldots,y_{u-1\,m},X_0). \\ &I_{u-1}''(y_{u-1\,1},\ldots,y_{u-1\,m},y_{u-2\,1},\ldots,y_{u-2\,m})\\ &\cap\, \mathbb{K}[y_{u-1\,1},\ldots,y_{u-1\,m}] \end{aligned} \tag{6.46}$$

By Theorem 6.26, we then obtain

$$\tilde{I}_*(y_{u-1\,1},\ldots,y_{u-1\,m},X_0) + \tilde{I}_u(x_1,\ldots,x_m,y_{u-1\,1},\ldots,y_{u-1\,m}) \cap \mathbb{K}[X] \subseteq I_*(X,X_0),$$

and thus, from (6.46), we have

$$\begin{aligned} & I'_1(y_{11},\ldots,y_{1m},X_0)+ \\ & I''_2(y_{21},\ldots,y_{2m},y_{11},\ldots,y_{1m})+ \\ & \cdots\ \cdots+ \qquad\qquad \subseteq I_*. \\ & I''_{u-1}(y_{u-1\ 1},\ldots,y_{u-1\ m},y_{u-2\ 1},\ldots,y_{u-2\ m})+ \\ & \tilde{I}_u(X,y_{u-1\ 1},\ldots,y_{u-1\ m})\cap\mathbb{K}[X] \end{aligned}$$

■

As presented already in Example 6.28, in Theorems 6.26 and 6.29 the inclusion of the elimination ideal of polynomial relations among the loop variables obtained from the ideal sum of the elimination ideals I'_1, $I''_2,\ldots,\tilde{I}''_k$ into the ideal I_* of polynomial relations among the loop variables obtained from the elimination on the merged closed form system is proper.

In Algorithm 6.19, *only* steps 1-4 of Algorithm 5.11 are used for computing the closed form system for each P-solvable inner loop from the loop sequence. However, in order to determine (all) polynomial relations among loop variables X with initial values X_0 for P-solvable loops with conditional branches and assignments, *the merging operation on closed form systems of P-solvable inner loops is a crucial and essential step in Algorithm 6.19.*

6.4 Automated Polynomial Invariant Generation by Algebraic Techniques

Let us fix some notation. Denote by J_* the ideal of invariant polynomial relations among the loop variables X with initial values X_0 for the P-solvable loop with $k \geq 1$ conditional branches and assignments. Based on Definition 6.11, J_* is thus the ideal of polynomial relations among the loop variables X with initial values X_0 that are satisfied by the initial values X_0 and are valid after arbitrary many iterations of $S_1,\ldots,S_k$ in any order, that is iterations of $S_1^* * \cdots * S_k^*$.

Hence, the conjunction of the polynomial invariant equations from *a basis of* J_* completely describe the polynomial loop invariants among the loop variables X with initial values X_0 for the P-solvable loop with k conditional branches and assignments.

Since by Definition 6.11, a polynomial invariant of the outer loop is a polynomial relation among the loop variables X with initial values X_0 that is valid at the entry of the outer loop and for any sequence of iterations of the outer loop, we proceed as follows.

(i) Note that the initial values X_0 of the loop variables X at the entry point of the outer loop are also the initial values of the loop variables X before the *first* iteration of the outer loop.

We thus firstly compute by Algorithm 6.22 the ideal of all polynomial relations among the loop variables X with initial values X_0 corresponding to the *first* iteration of the outer loop. We denote this ideal by PI_1.

(ii) Next, based on Definition 6.11, from (the generators of) PI_1 we keep only the set GI of those polynomial relations that are polynomial invariants among the loop variables X with initial values X_0: they are preserved by any iteration of the outer loop starting in a state in which the initial values of the loop variables X are X_0.

By correctness of Theorem 6.6, the polynomials from *GI* thus obtained are polynomial invariants among the loop variables X with initial values X_0 of the P-solvable loop with k conditional branches and assignments (see Theorem 6.34).

Further, under *additional assumptions* from Subsection 6.5, we prove that our algorithm is complete. Namely, *GI* generates the ideal of polynomial invariants among the loop variables X with initial values X_0 for some special cases of P-solvable loops with conditional branches and assignments .

In more detail, we proceed as follows.

Step (i).

As mentioned already, the initial values X_0 of the loop variables X at the entry point of the outer loop are the initial values of the loop variables X before the *first* iteration of the outer loop.

Hence, step (i) is a direct consequence of Theorem 6.23.

We denote

$$PI_1 := \text{Algorithm 6.22}(S_1^{j_1}, \ldots, S_k^{j_k}). \tag{6.47}$$

Example 6.30 Assuming that $a[0], b[0], d[0], y[0]$ denote the initial values before the *first* iteration of the outer loop (6.7), from Example 6.25 we thus derive

$$\begin{aligned} PI_1 = \quad & \langle\, -b[0]*d + b*d[0], \\ & a*d[0] - a[0]*d[0] - 2*b[0]*y + 2*b[0]*y[0], \\ & a*d - a[0]*d - 2*b*y + 2*b*y[0] \rangle. \end{aligned}$$

By correctness of Algorithm 6.22, PI_1 is thus the ideal of all valid polynomial relations among the loop variables X with initial values X_0 after the *first* iteration of the outer loop. Hence, for all $p \in PI_1$ we have

$$\{p(X) = 0 \wedge X = X_0\} S_1^* * \ldots * S_n^* \{p(X) = 0\}.$$

Based on Remark 6.10 and (6.21) we have that *any polynomial invariant* among the loop variables X with initial values X_0 of the P-solvable loop with k conditional branches and assignments is also *a valid polynomial relation for the first iteration* of the P-solvable loop. Using the introduced notation, we thus have

$$J_* \subseteq PI_1. \tag{6.48}$$

However, not all polynomial relations from PI_1 are also polynomial invariants for the P-solvable loop. In other words, the inclusion $PI_1 \subseteq J_*$ *does not hold* in general (see e.g. Example 6.43).

In order to obtain polynomial invariants of the outer loop, we thus need a systematic method to keep only those polynomials from the ideal PI_1 that are invariants of the outer loop, and thus are in J_*. This is presented below.

Step (ii).

From (a basis of) PI_1 computed by Algorithm 6.22 we determine the set GI of all those polynomial relations from PI_1 that are valid for each $S_1,\ldots,S_k$. Hence, the *conjunction of the polynomials from GI* is preserved by each $S_1,\ldots,S_k$.

W.l.o.g., by any operation over the ideals PI_1 we mean algorithmic manipulation over a (Gröbner) basis of PI_1.

The branches S_i of conditional statements are sequences of assignments. As mentioned already, in case of assignments, wp commutes with the logical connectives. Thus, checking whether a conjunction of polynomials $p(X)$ from GI is preserved by each S_i reduces to the following procedure.

(1) Compute the weakest precondition $\mathrm{wp}(S_i, p(X) = 0)$ for S_i and the (postcondition corresponding to the) polynomial relation $p(X) = 0$.

(2) Check whether the polynomial relation among the loop variables X corresponding to the polynomial equation $\mathrm{wp}(S_i, p(X) = 0)$ is in the ideal generated by GI, that is an ideal membership problem and can be algorithmically answered by Gröbner basis computation (see Remark 5.6 and Theorem 4.10).

In what follows, w.l.o.g., by $\mathrm{wp}(S_i, p(X) = 0) \in \langle GI\rangle$ we understand the ideal ($\langle GI\rangle$) membership of the polynomial relation among the loop variables X corresponding to the polynomial equation $\mathrm{wp}(S_i, p(X) = 0)$.

We have thus the following theorem.

Theorem 6.31

Let

$$GI = \{p(X) \in PI_1 \mid wp(S_i, p(X) = 0) \in \langle GI\rangle, i = 1,\ldots,k\} \subseteq PI_1. \tag{6.49}$$

Then

$$GI \subseteq J_*. \tag{6.50}$$

Namely, any polynomial from GI is a polynomial invariant among the loop variables X with initial values X_0 of the P-solvable loop with k conditional branches and assignments.

Proof. Since $GI \subseteq PI_1$, by correctness of Algorithm 6.22 we have

$$\forall p \in GI: \qquad \{p(X) = 0 \wedge X = X_0\} \quad S_1^* | \ldots | S_k^* \quad \{p(X) = 0\},$$

yielding that

$$\{\bigwedge_{p\in GI} p(X) = 0 \wedge X = X_0\} \quad S_1^* | \ldots | S_k^* \quad \{\bigwedge_{p\in GI} p(X) = 0\}. \tag{6.51}$$

Moreover, by (6.49) and Remark 5.6, we obtain

$$\forall i = 1,\ldots,k: \qquad \{\bigwedge_{p\in GI} p(X) = 0\} \quad S_i^* \quad \{\bigwedge_{p\in GI} p(X) = 0\},$$

which, by the definition of $|$, yields

$$\{\bigwedge_{p\in GI} p(X) = 0\} \quad S_1^* | \ldots | S_k^* \quad \{\bigwedge_{p\in GI} p(X) = 0\}. \tag{6.52}$$

Further, using Remark 6.10, from (6.51) and (6.52) we derive

$$\{\bigwedge_{p\in GI} p(X)=0 \wedge X=X_0\} \quad (S_1|\ldots|S_k)^* \quad \{\bigwedge_{p\in GI} p(X)=0\}. \tag{6.53}$$

Finally, by Definitions 6.11 and 5.8, from (6.53) we conclude that $\bigwedge_{p\in GI} p(X)=0$ is a polynomial loop invariant, and thus

$$GI \subseteq J_*.$$

■

Finally, we can now formulate our algorithm for polynomial invariant generation for P-solvable loops with conditional branches and assignments.

Algorithm 6.32 (P-solvable Loops with Conditionals)
Input: Imperative P-solvable loop with $k \geq 1$ conditional branches and assignments
Output: Polynomial invariants among the loop variables X with initial values X_0 of the P-solvable loop
Assumption: S_i, $i = 1,\ldots,k$, are sequences of assignments from one conditional branch

1 By Theorem 6.6, transform the loop into a nested loop system with $k \geq 1$ P-solvable inner loops $S_1,\ldots,S_k$

2 Apply Algorithm 6.22 for determining the ideal PI_1 of polynomial relations among the loop variables after the first iteration of the outer loop

3 From PI_1 keep the set GI of those polynomials whose conjunction is preserved by $S_1,\ldots,S_k$:

$GI = \{p \in PI_1 \mid wp(S_i, p(X)=0) \in \langle GI\rangle, i=1,\ldots,k\} \subset PI_1$

where $wp(S_i, p(X)=0)$ is the weakest precondition for S_i and the postcondition $p(X)=0$

4 *return* GI.

At step 3 of Algorithm 6.32, the set GI is determined using a *(Gröbner) basis* of PI_1. More precisely, step 3 of Algorithm 6.32 is computed by Algorithm 6.33 as follows.

Algorithm 6.33 "receives" as input a set B_1 of polynomial relations among the loop variables X with initial values X_0 representing a basis for the ideal PI_1 for the first iteration of the P-solvable loop with k conditional branches and assignments. B_1 is thus computed by Algorithm 6.22 with the initial values X_0, Next, from B_1 we keep only the set GI of those polynomial relations that are preserved by each inner loop S_i. For doing so, we first initialize GI to B_1 (step 1 of Algorithm 6.33). Further,

- the weakest preconditions for S_i, $i = 1,\ldots,k$, and polynomial equation (i.e. postcondition) corresponding to any polynomial from $p \in GI$ are first computed,

- and then check whether the weakest preconditions thus derived are in the ideal generated by GI.

That is an ideal membership problem, and hence we check whether the weakest preconditions can be reduced to 0 w.r.t. (a subset of) GI (step 5 of Algorithm 6.33). If affirmative, the polynomials from GI used in the reduction process of the weakest preconditions are kept in DP (step 6 of

Algorithm 6.33). Otherwise, the polynomial p is not preserved by each S_i, and thus it is not in GI (step 8 of Algorithm 6.33).

Intuitively, GI is the set of those polynomials from PI_1 for which the weakest preconditions of S_i, $i = 1,\ldots,k$, are reduced to 0 w.r.t. GI, whereas DP stands for the set of dependency polynomials of GI, namely, the set of polynomials used in the reduction process of the weakest preconditions.

We iterate this process until the set of polynomials used in the reduction process of the weakest preconditions is a subset of GI (step 11 of Algorithm 6.33). Thus, any polynomial from GI is preserved by each S_i, namely, the weakest preconditions for S_i and polynomial equations corresponding to any polynomial $p \in GI$ can be reduced w.r.t. GI.

Algorithm 6.33 (Polynomial Invariant Filtering on PI_1)
Input: Set $B_1 = \{p_1,\ldots,p_r\}$ consisting of valid polynomial relations among the loop variables X with initial values X_0 for the first iteration of a P-solvable loop with k conditional branches and assignments, such that $\langle B_1\rangle = PI_1$
Output: Set $GI \subseteq B_1$ of polynomial invariants among the loop variables X with initial values X_0 of the P-solvable loop
Assumption: S_i, $i = 1,\ldots,k$, are sequences of assignments from the conditional branches

1 $GI = B_1$
2 **repeat**
3 $DP = \emptyset$
4 **for each** $p \in GI$ **do**
5 **if** $\Big(\mathrm{wp}(S_1, p(X)=0)_{\xrightarrow[\text{reduced w.r.t. } A_1\subseteq GI]{}} 0\Big) \wedge \cdots \wedge \Big(\mathrm{wp}(S_k, p(X)=0)_{\xrightarrow[\text{reduced w.r.t. } A_k\subseteq GI]{}} 0\Big)$ **then**
6 $DP = DP \cup \Big(\bigcup_{i=1,\ldots,k} A_i\Big)$
7 **else**
8 $GI = GI \setminus \{p\}$
9 **endif**
10 **endfor**
11 **until** $DP \subseteq GI$
12 **return** GI.

We thus have the theorem below.

Theorem 6.34 Algorithm 6.32 is correct. It returns polynomial invariants among the loop variables X with initial values X_0 of the P-solvable loop with k conditional branches and assignments.

Proof. It follows from the correctness of Algorithm 6.22 and Theorem 6.31. ∎

Example 6.35 Denoting by $B_1 = \{b[0]*d + b*d[0], a*d[0] - a[0]*d[0] - 2*b[0]*y + 2*b[0]*y[0], a*d - a[0]*d - 2*b*y + 2*b*y[0]\}$, by Example 6.30 we have $PI_1 = \langle B_1\rangle$.
For simplicity, let us denote

$$p_1(a,b,d) = b[0]*d + b*d[0],$$
$$p_2(a,b,d) = a*d[0] - a[0]*d[0] - 2*b[0]*y + 2*b[0]*y[0],$$
$$p_3(a,b,d) = a*d - a[0]*d - 2*b*y + 2*b*y[0],$$

and we initialize $GI = B_1$.

Next, we compute the weakest preconditions of S_1 and S_2 w.r.t. the polynomial equations corresponding to each $p_i,\ i = 1,2,3\text{m}$ from B_1. (According to Example 6.2, we have $S_1 = b := b/2; d := d/2$ and $S_2 = a := a+b; y := y+d/2; b := b/2; d := d/2$.)

We then have

$$\begin{aligned}
\text{wp}\big(S_1,\ p_1(a,b,d)=0\big) &\equiv \quad 1/2 * p_1(a,b,d) = 0,\\
\text{wp}\big(S_2,\ p_1(a,b,d)=0\big) &\equiv \quad 1/2 * p_1(a,b,d) = 0,\\
\\
\text{wp}\big(S_1,\ p_2(a,b,d)=0\big) &\equiv \quad p_2(a,b,d) = 0,\\
\text{wp}\big(S_2,\ p_2(a,b,d)=0\big) &\equiv \quad -p_1(a,b,d) + p_2(a,b,d) = 0,\\
\\
\text{wp}\big(S_1,\ p_3(a,b,d)=0\big) &\equiv \quad 1/2 * p_3(a,b,d) = 0,\\
\text{wp}\big(S_2,\ p_3(a,b,d)=0\big) &\equiv \quad 1/2 * p_3(a,b,d) = 0.
\end{aligned}$$

Since

$$\begin{aligned}
&1/2 * p_1(a,b,d) && \xrightarrow[\text{reduced w.r.t. } \{p_1\}\subseteq GI]{} && 0,\\
&p_2(a,b,d) && \xrightarrow[\text{reduced w.r.t. } \{p_2\}\subseteq GI]{} && 0,\\
&-p_1(a,b,d) + p_2(a,b,d) && \xrightarrow[\text{reduced w.r.t. } \{p_1,p_2\}\subseteq GI]{} && 0,\\
&1/2 * p_3(a,b,d) && \xrightarrow[\text{reduced w.r.t. } \{p_3\}\subseteq GI]{} && 0,
\end{aligned}$$

we derive

$$\begin{aligned}
GI = &\quad \{p_1, p_2, p_3\},\\
DP = &\quad \{p_1, p_2, p_3\},
\end{aligned}$$

We thus have

$$GI = DP = B_1.$$

Hence, the set of polynomial invariants obtained by Algorithm 6.32 is

$$\begin{aligned}
GI = \quad &\{b[0] * d + b * d[0],\\
&\ a * d[0] - a[0] * d[0] - 2 * b[0] * y + 2 * b[0] * y[0],\\
&\ a * d - a[0] * d - 2 * b * y + 2 * b * y[0]\}.
\end{aligned}$$

Thus, our algorithm for invariant generation combines algorithmic combinatorics with polynomial algebra such that it returns polynomial invariants among the loop variables with initial values X_0 of the P-solvable loop with conditional branches and assignments.

Moreover, in the next subsection, we prove that *under additional assumptions*, our algorithm for invariant generation is complete. Namely, GI is the ideal of *all* polynomial invariants among the loop variables with initial values X_0 of some special cases of P-solvable loops with conditional branches and assignments.

6.5 Some Completeness Results

Based on Section 5, note that for $k = 1$, i.e. no conditional branches in the loop body, our approach is complete in generating a basis for the polynomial invariant ideal.

In this section we present under which assumptions our polynomial invariant generation algorithm for P-solvable loops with $k \geq 1$ conditional branches and assignments is complete.

However, all examples treated in the thesis fulfill these assumptions, yielding thus the completeness of our approach. Namely, for all examples in the thesis, Algorithm 6.32 returns a basis for the polynomial invariant ideal .

Moreover, we could not find any example for which the additional assumptions introduced below are violated. We thus conjecture that our method is complete for a rich class of P-solvable loops with conditional branches and assignments. The study of completeness of our approach in the general case (i.e. without the imposed constraints) remains a challenging task for further research.

We keep the notation of J_* as introduced in Subsection 6.4. Namely, J_* is the ideal of all polynomial invariants among the loop variables X with initial values X_0 of the P-solvable loop with k conditional branches and assignments.

For proving completeness of our method, we impose structural conditions on the ideal of polynomial relations among the loop variables X with initial values X_0 corresponding to sequences of k and $k+1$ inner loops, as presented below.

Let us first treat the *base case of* $k = 2$.

The considered P-solvable loops are thus with 2 conditional branches and assignments, with all tests being ignored. Namely, we have P-solvable loops with non-deterministic branches as presented below.

```
While[...,
    If[...,S1]
    []
    If[...,S2]].
```

As mentioned before, S_1 and S_2 are sequences of assignments.

Let us fix some further notation.

- J_1 denotes the ideal of polynomial relations among the loop variables X after arbitrary $j_1 \in \mathbb{N}$ and $j_2 \in \mathbb{N}$ iterations of S_1 and S_2 respectively, i.e. after $\underbrace{S_1;\ldots;S_1}_{j_1 \text{ times}};\underbrace{S_2;\ldots;S_2}_{j_2 \text{ times}}$, parameterized by the initial values X_0. In other words, J_1 is the ideal of valid polynomial relations among the loop variables X with initial values X_0 corresponding to the sequence of assignments $S_1^{j_1};S_2^{j_2}$.

  J_1 is thus computed by Algorithm 6.19, namely:

  $$J_1 = \text{Algorithm 6.19}(S_1^{j_1}, S_2^{j_2}).$$

  Based on correctness of Algorithm 6.19 and (6.34), we thus have

  $$\forall j_1, j_2 \in \mathbb{N}: \qquad \{p(X) = 0 \wedge X = X_0\} \quad S_1^{j_1};S_2^{j_2} \quad \{p(X) = 0\}, \tag{6.54}$$

which, using Remark 6.10, can be equivalently written as

$$\{p(X) = 0 \wedge X = X_0\} \quad S_1^*; S_2^* \quad \{p(X) = 0\}. \tag{6.55}$$

- J_2 denotes the ideal of polynomial relations among the loop variables X after arbitrary $j_2 \in \mathbb{N}$ and $j_1 \in \mathbb{N}$ iterations of S_2 and S_1 respectively, i.e. after $\underbrace{S_2;\ldots;S_2}_{j_2 \text{ times}};\underbrace{S_1;\ldots;S_1}_{j_1 \text{ times}}$, parameterized by the initial values X_0. In other words, J_2 is the ideal of valid polynomial relations among the loop variables X with initial values X_0 corresponding to the sequence of assignments $S_2^{j_2}; S_1^{j_1}$.

  J_2 is thus computed by Algorithm 6.19, namely:

  $$J_2 = \text{Algorithm 6.19}(S_2^{j_2}, S_1^{j_1}).$$

  Based on correctness of Algorithm 6.19 and (6.34), we thus have

  $$\forall j_2, j_1 \in \mathbb{N}: \qquad \{p(X) = 0 \wedge X = X_0\} \quad S_2^{j_2}; S_1^{j_1} \quad \{p(X) = 0\}, \tag{6.56}$$

  that, using Remark 6.10, can be equivalently written as

  $$\{p(X) = 0 \wedge X = X_0\} \quad S_2^*; S_1^* \quad \{p(X) = 0\}. \tag{6.57}$$

- J_3 denotes the ideal of polynomial relations among the loop variables X after arbitrary $j_1 \in \mathbb{N}$, $j_2 \in \mathbb{N}$ and $j_3 \in \mathbb{N}$ iterations of S_1, S_2 and S_1 respectively, i.e. after $\underbrace{S_1;\ldots;S_1}_{j_1 \text{ times}};\underbrace{S_2;\ldots;S_2}_{j_2 \text{ times}};\underbrace{S_1;\ldots;S_1}_{j_3 \text{ times}}$, parameterized by the initial values X_0. In other words, J_3 is the ideal of valid polynomial relations among the loop variables X with initial values X_0 corresponding to the sequence of assignments $S_1^{j_1}; S_2^{j_2}; S_1^{j_3}$.

  J_3 is thus computed by Algorithm 6.19, namely:

  $$J_3 = \text{Algorithm 6.19}(S_1^{j_1}, S_2^{j_2}, S_1^{j_3}).$$

  Based on correctness of Algorithm 6.19 and (6.34), we thus have

  $$\forall j_1, j_2, j_3 \in \mathbb{N}: \qquad \{p(X) = 0 \wedge X = X_0\} \quad S_1^{j_1}; S_2^{j_2}; S_1^{j_3} \quad \{p(X) = 0\}, \tag{6.58}$$

  that, using Remark 6.10, can be equivalently written as

  $$\{p(X) = 0 \wedge X = X_0\} \quad S_1^*; S_2^*; S_1^* \quad \{p(X) = 0\}. \tag{6.59}$$

- J_4 denotes the ideal of polynomial relations among the loop variables X after arbitrary $j_2 \in \mathbb{N}$, $j_1 \in \mathbb{N}$ and $j_4 \in \mathbb{N}$ iterations of S_2, S_1 and S_2 respectively, i.e. after $\underbrace{S_2;\ldots;S_2}_{j_2 \text{ times}};\underbrace{S_1;\ldots;S_1}_{j_1 \text{ times}};$

$\underbrace{S_2;\ldots;S_2}_{j_4 \text{ times}}$, parameterized by the initial values X_0. In other words, J_4 is the ideal of valid polynomial relations among the loop variables X with initial values X_0 corresponding to the sequence of assignments $S_2^{j_2};S_1^{j_1};S_2^{j_4}$.

J_4 is thus computed by Algorithm 6.19, namely:

$$J_4 = \text{Algorithm 6.19}(S_2^{j_2}, S_1^{j_1}, S_2^{j_4}).$$

Based on correctness of Algorithm 6.19 and (6.34), we thus have

$$\forall j_2, j_1, j_4 \in \mathbb{N}: \qquad \{p(X) = 0 \wedge X = X_0\} \quad S_2^{j_2};S_1^{j_1};S_2^{j_4} \quad \{p(X) = 0\}, \tag{6.60}$$

that, using Remark 6.10, can be equivalently written as

$$\{p(X) = 0 \wedge X = X_0\} \quad S_2^*;S_1^*;S_2^* \quad \{p(X) = 0\}. \tag{6.61}$$

Note, that any polynomial relation from J_3 is in J_1, since the corresponding assignment sequences for which J_1 describes their ideal of polynomial relations represent a subclass (smaller set) of assignment sequences for which J_3 represents their ideal of polynomial relations. In other words, $S_1^{j_1};S_2^{j_2};S_3^{j_3}$ describes a wider set of programs than $S_1^{j_1};S_2^{j_2}$. Therefore any valid polynomial relation among the loop variables X with initial values X_0 of $S_1^{j_1};S_2^{j_2};S_3^{j_3}$ is also a valid polynomial relation among the loop variables X with initial values X_0 of the special case $S_1^{j_1};S_2^{j_2}$.

Similarly, any polynomial relation from J_4 is in J_2.

Moreover, any polynomial relation from the ideal J_* of polynomial invariants among the loop variables X with initial values X_0 for the P-solvable loop is obviously in J_1, J_2, J_3 and J_4.

We thus have

$$J_* \subseteq J_1, \qquad J_* \subseteq J_2, \qquad J_* \subseteq J_3, \qquad J_* \subseteq J_4. \tag{6.62}$$

However, in order to completely characterize (a basis of) J_* these inclusions are not enough.

In what follows we formulate constraints on the ideal of polynomial relations J_1, J_2, J_3 and J_4, under which assumptions J_* can be completely obtained from J_1, J_2, J_3 and J_4.

The theorems below impose structural constraints on the ideals J_1, J_2, J_3 and J_4, such that our algorithm for invariant generation is complete for P-solvable loops with $k = 2$ conditional branches.

Theorem 6.36 If

$$J_1 \cap J_2 = J_3 \cap J_4 \tag{6.63}$$

then

$$J_* = J_1 \cap J_2.$$

Proof. Let $\mathfrak{a} = J_1 \cap J_2$. By assumption (6.63) we have

$$\mathfrak{a} = J_1 \cap J_2 = J_3 \cap J_4.$$

Hence, the polynomial relations from (a basis of) $\mathfrak{a}$ are all polynomial relations among the loop variables X with initial values X_0 that are valid for $S_1^{j_1};S_2^{j_2}$, $S_2^{j_2};S_1^{j_1}$, $S_1^{j_1};S_2^{j_2};S_1^{j_3}$ and $S_2^{j_2};S_1^{j_1};S_2^{j_4}$, i.e. they are in the intersection ideal of J_1, J_2, J_3 and J_4.

By (6.55), (6.57), (6.59) and (6.61) we thus derive

$$\forall p \in \mathfrak{a}: \quad \{p(X) = 0 \wedge X = X_0\} \quad S_1^*;S_2^* \quad \{p(X) = 0\}, \tag{6.64}$$
$$\forall p \in \mathfrak{a}: \quad \{p(X) = 0 \wedge X = X_0\} \quad S_2^*;S_1^* \quad \{p(X) = 0\}, \tag{6.65}$$
$$\forall p \in \mathfrak{a}: \quad \{p(X) = 0 \wedge X = X_0\} \quad S_1^*;S_2^*;S_1^* \quad \{p(X) = 0\}, \tag{6.66}$$
$$\forall p \in \mathfrak{a}: \quad \{p(X) = 0 \wedge X = X_0\} \quad S_2^*;S_1^*;S_2^* \quad \{p(X) = 0\}. \tag{6.67}$$

Based on the sequencing rule of Hoare-logic calculus, from (6.64) and (6.66), we obtain the formula

$$\forall p \in \mathfrak{a}: \quad \{p(X) = 0\} \quad S_1^* \quad \{p(X) = 0\}. \tag{6.68}$$

Similarly, from (6.65) and (6.67), we obtain

$$\forall p \in \mathfrak{a}: \quad \{p(X) = 0\} \quad S_2^* \quad \{p(X) = 0\}. \tag{6.69}$$

Using again the sequencing rule of Hoare-logic calculus, by composing (6.66) and (6.67) with (6.68) and (6.69) arbitrary many times, we then have

$$\forall p \in \mathfrak{a}: \quad \{p(X) = 0 \wedge X = X_0\} \quad (S_1^*|S_2^*)^* \quad \{p(X) = 0\},$$
$$\forall p \in \mathfrak{a}: \quad \{p(X) = 0 \wedge X = X_0\} \quad (S_2^*|S_1^*)^* \quad \{p(X) = 0\},$$

that, by Remark 6.10, can be equivalently written as

$$\forall p \in \mathfrak{a}: \quad \{p(X) = 0 \wedge X = X_0\} \quad (S_1|S_2)^* \quad \{p(X) = 0\}. \tag{6.70}$$

Using Definition 6.11, we thus obtain

$$\forall p \in \mathfrak{a}: \quad p \in J_*.$$

Hence

$$\mathfrak{a} \subseteq J_*,$$

and by (6.62) we finally have

$$J_* = J_1 \cap J_2.$$

■

By correctness of Theorem 6.1, note that the intersection ideal $\mathfrak{a} = J_1 \cap J_2$ is the ideal of all polynomial relations among the loop variables X with initial values X_0 corresponding to the *first* iteration of the outer loop having the 2 inner loops S_1 and S_2 with j_1 and j_2 iterations respectively. Based on Theorem 6.23, we thus have

$$\mathfrak{a} = \text{Algorithm 6.22}(S_1^*, S_2^*).$$

Hence, by construction of Algorithm 6.32, $\mathfrak{a} = PI_1$ and it is computed at step 2 of Algorithm 6.32.

The next theorem imposes structural assumptions on the output *GI* of Algorithm 6.32 in order to establish a relation between *GI*, $\mathfrak{a}$ and J_* from which completeness of our algorithm can be derived.

Theorem 6.37 Let *GI* be as considered in Algorithm 6.32.
If

$$\langle GI \rangle = J_1 \cap J_2 \cap J_3 \cap J_4 \tag{6.71}$$

then

$$J_* = \langle GI \rangle = J_1 \cap J_2 \cap J_3 \cap J_4.$$

Proof. By (6.62), we have

$$J_* \subseteq J_1 \cap J_2 \cap J_3 \cap J_4. \tag{6.72}$$

Further, by Theorem 6.31 and correctness of Algorithm 6.32 we have

$$\langle GI \rangle \subseteq J_*, \tag{6.73}$$

and by (6.72) we get the chain of inclusions

$$\langle GI \rangle \subseteq J_* \subseteq J_1 \cap J_2 \cap J_3 \cap J_4,$$

from which, using the assumption (6.71), we finally obtain

$$J_* = \langle GI \rangle = J_1 \cap J_2 \cap J_3 \cap J_4.$$

■

A stronger constraint for proving the completeness of our invariant generation approach presented in Algorithm 6.32 is given by the theorem below.

Theorem 6.38 Let *GI* be as considered in Algorithm 6.32.
If

$$\langle GI \rangle = J_1 \cap J_2 \tag{6.74}$$

then

$$J_* = \langle GI \rangle = J_1 \cap J_2.$$

Proof. The proof is similar to the proof of Theorem 6.37.
By (6.62), we have

$$J_* \subseteq J_1 \cap J_2. \tag{6.75}$$

Further, by Theorem 6.31 and correctness of Algorithm 6.32 we have

$$\langle GI \rangle \subseteq J_*, \tag{6.76}$$

and by (6.75) we get the chain of inclusions

$$\langle GI \rangle \subseteq J_* \subseteq J_1 \cap J_2,$$

from which, using the assumption (6.74), we finally obtain

$$J_* = \langle GI \rangle = J_1 \cap J_2.$$

■

Note that if (6.74) holds, then (6.71) and (6.63) are also fulfilled. In other words, Theorem 6.38 implies the validity of Theorems 6.37 and 6.36, as it is presented below.

- Algorithm 6.32 and (6.62) implies the chain of inclusions

$$\langle GI \rangle \subseteq J_* \subseteq J_1 \cap J_2 \cap J_3 \cap J_4 \subseteq J_3 \cap J_4 \subseteq J_1 \cap J_2.$$

- By (6.74), we thus obtain
  (1) $\langle GI \rangle = J_1 \cap J_2 \cap J_3 \cap J_4$, yielding Theorem 6.37;
  (2) $J_3 \cap J_4 = J_1 \cap J_2$, yielding Theorem 6.36.

Hence, in the completeness study of our approach, we first check the validity of (6.74). If it is fulfilled, we conclude that Algorithm 6.32 returns a complete set of polynomial invariants from which any further invariant of the loop can be derived. Otherwise, Theorems 6.37 and 6.36 are verified in order to check the completeness of our invariant generation algorithm for P-solvable loops with $k = 2$ conditional branches and assignments.

Example 6.39 From Examples 6.30 and 6.35 we obtain

$$GI = PI_1.$$

By Theorem 6.38 we thus derive that

$$GI = J_*,$$

yielding the completeness of Algorithm 6.32 for Example 6.2.

For simplicity, in Theorems 6.36, 6.37 and 6.38 we first considered the case of P-solvable loops with only $k = 2$ conditional branches. However, these results can be easily extended to the *general case of P-solvable loops* (6.20) *with* $k \geq 1$ *conditional branches and assignment statements*.

As mentioned already, in the general case of having a P-solvable loop with $k \geq 2$ conditional branches with ignored test conditions, by applying Theorem 6.6, we obtain an outer loop with k P-solvable inner loops $S_1, \ldots, S_k$ with only assignments. Based on Remark 6.13, for any iteration of the outer loop we have $k!$ possible sequences of inner P-solvable loops.

Consider an arbitrary permutation $W = (w_1, \ldots, w_k) \in \mathfrak{S}_k$, and an arbitrary sequence of numbers $J = \{j_{w_1}, \ldots, j_{w_k}\} \in \mathbb{N}^k$ to denote an arbitrary iteration $S_W^J = S_{w_1}^{j_{w_1}}; \ldots; S_{w_k}^{j_{w_k}}$ of the outer loop.

Let us first fix some further notation.

- J_W denotes the ideal of valid polynomial relations among the loop variables X with initial values X_0 after S_W^J, i.e. after $\underbrace{S_{w_1}; \ldots; S_{w_1}}_{j_{w_1} \text{ times}}; \ldots; \underbrace{S_{w_k}; \ldots; S_{w_k}}_{j_{w_k} \text{ times}}$.

  J_W is thus computed by Algorithm 6.19, namely:

$$J_W = \text{Algorithm 6.19}(S_{w_1}^{j_{w_1}}, \ldots, S_{w_k}^{j_{w_k}}).$$

Based on correctness of Algorithm 6.19 and (6.34), we thus have

$$\forall j_{w_1},\ldots,j_{w_k}\in\mathbb{N}: \qquad \{p(X)=0\wedge X=X_0\} \quad S_{w_1}^{j_{w_1}};\ldots;S_{w_k}^{j_{w_k}} \quad \{p(X)=0\}, \tag{6.77}$$

that, using Remark 6.10, can be equivalently written as

$$\{p(X)=0\wedge X=X_0\} \quad S_{w_1}^*;\ldots;S_{w_k}^* \quad \{p(X)=0\}. \tag{6.78}$$

- For all $i=1,\ldots,k$ and $j\in\mathbb{N}$, we denote by $J_{W,i}$ the ideal of valid polynomial relations among the loop variables X with initial values X_0 after $S_W^J;S_i^j$. Namely, $J_{W,i}$ is the ideal of all valid polynomial relations after $\underbrace{S_{w_1};\ldots;S_{w_1}}_{j_{w_1}\text{ times}};\ldots;\underbrace{S_{w_k};\ldots;S_{w_k}}_{j_{w_k}\text{ times}};\underbrace{S_i;\ldots;S_i}_{j\text{ times}}$.

  $J_{W,i}$ is thus computed by Algorithm 6.19 on a sequence of $k+1$ inner loops as follows.

$$J_{W,i}=\text{Algorithm 6.19}(S_{w_1}^{j_{w_1}},\ldots,S_{w_k}^{j_{w_k}},S_i^j).$$

  Based on correctness of Algorithm 6.19 and (6.34), we thus have

$$\forall j_{w_1},\ldots,j_{w_k},j\in\mathbb{N}: \qquad \{p(X)=0\wedge X=X_0\} \quad S_{w_1}^{j_{w_1}};\ldots;S_{w_k}^{j_{w_k}};S_i^j \quad \{p(X)=0\}, \tag{6.79}$$

  that, using Remark 6.10, can be equivalently written as

$$\{p(X)=0\wedge X=X_0\} \quad S_{w_1}^*;\ldots;S_{w_k}^*;S_i^* \quad \{p(X)=0\}. \tag{6.80}$$

The theorem below addresses the completeness aspect of our invariant generation algorithm for P-solvable loops (6.20).

Theorem 6.40

Let *GI* be as considered in Algorithm 6.32 for P-solvable loops with $k\geq 1$ conditional branches and assignments.

Let $\mathfrak{a}_k=\bigcap_{W\in\mathfrak{S}_k}J_W$ and $\mathfrak{a}_{k+1}=\bigcap_{\substack{W\in\mathfrak{S}_k\\ i=1,\ldots,k}}J_{W,i}$.

(1) If $\mathfrak{a}_k=\mathfrak{a}_{k+1}$ then $J_*=\mathfrak{a}_k$.

(2) If $\langle GI\rangle=\mathfrak{a}_k\cap\mathfrak{a}_{k+1}$ then $J_*=\mathfrak{a}_k\cap\mathfrak{a}_{k+1}$.

(3) If $\langle GI\rangle=\mathfrak{a}_k$ then $J_*=\mathfrak{a}_k$.

Proof. It follows from a similar reasoning as for Theorems 6.36, 6.37 and 6.38, based on Hoare-logic calculus and Definition 6.11.

■

By correctness of Theorem 6.6, note that the intersection ideal $\mathfrak{a}_k$ is the ideal of all polynomial relations among the loop variables X with initial values X_0 corresponding to the *first* iteration of the outer loop having the k inner loops $S_1,\ldots,S_k$ with $j_1,\ldots,j_k$ iterations respectively. Based on Theorem 6.23, we then have

$$\mathfrak{a}=\text{Algorithm 6.22}(S_1^*,\ldots,S_k^*).$$

Hence, $\mathfrak{a}_k = PI_1$ and it is computed at step 2 of Algorithm 6.32.

However, Theorem 6.40 increases the computation of ideals among polynomial relations. For the general case of k inner loops, since $\mathfrak{a}_k$ is the intersection ideal of $k!$ ideals, $\mathfrak{a}_{k+1}$ requires computation of $(k-1)k!$ ideals. Note that for $k = 1$, $\mathfrak{a}_{k+1}$ is not computed, the completeness of our approach for $k = 1$ always holds as mentioned in Section 5.

Under the assumptions from Theorem 6.40, our method is thus complete in generating the ideal of all polynomial invariants among the loop variables X with initial values X_0 for the P-solvable loop with $k \geq 1$ conditional branches and assignments.

As mentioned already, the completeness of our polynomial invariant generation algorithm is satisfied for all examples treated in the thesis. Moreover, we could not find any example for which our method fails to be complete, and we thus conjecture that Subsection 6.5 covers a rich class of P-solvable loops with k conditional branches and assignments.

A Decidable Class: P-solvable Loops with Conditional Branches and Affine Assignments

As discussed already, any closed form solution of a loop variable defined by an affine relation is a linear combination of polynomials and algebraically related exponentials in the summation variable (i.e. loop counter).

In Subsection 5.6, for the case of P-solvable loops with only assignments, we proved that any affine loop is P-solvable loop. Thus, affine loops are treated in a complete manner by our invariant generation for P-solvable loops with assignments only.

Going further, based on results from Subsections 6.4 and 6.5, we can generalize the results of Subsection 5.6 by considering P-solvable loops with *conditional branches and affine assignments*. For such loops, by Theorem 6.6, we obtain a nested loop system with affine loops.

Corollary 6.41 Given an imperative loop as in (6.20) having only conditional branches and affine assignments, with the property that test conditions are ignored.

Then the polynomial relations among the loop variables X with initial values X_0 returned by Algorithm 6.32 are polynomial invariants of the imperative loop with conditional branches and affine assignments.

Proof. It follows from Theorem 5.20 and from the correctness of Algorithm 6.32.

■

Moreover, under the imposed structural restrictions from Subsection 6.5, Corollary 6.41 yields the completeness of our approach. Namely, it returns the ideal of all polynomial invariants for an imperative loop as in (6.20) having only conditional branches and affine assignments, with ignored test conditions in the loop and conditional branches.

Example 6.42 Given the imperative loop with conditional branches and assignments:

$$
\begin{array}{ll}
 & r := a-1;\ q := 1;\ p := 1/2; \\
 & \texttt{While}[(2*p*r \geq err), \\
 & \texttt{If}[2*r-2*q*p \geq 0 \\
S_1: & \quad \texttt{Then}\ r := 2*r-2*q-p;\ q := q+p;\ p := p/2 \\
S_2: & \quad \texttt{Else}\ r := 2*r;\ p := p/2].
\end{array}
$$

By applying Algorithm 6.32 and using our software package `Aligator`, the polynomial invariants are obtained as follows.

Step 1: Loop transformation

$$
\begin{array}{ll}
 & \texttt{While}[(2*p*r \geq err), \\
 & \quad \texttt{While}[(2*p*r \geq err) \wedge (2*r-2*q*p \geq 0), \\
S_1: & \quad\quad r := 2*r-2*q-p;\ q := q+p;\ p := p/2]; \\
 & \quad \texttt{While}[(2*p*r \geq err) \wedge \neg(2*r-2*q*p \geq 0) \\
S_2: & \quad\quad r := 2*r;\ p := p/2]].
\end{array}
$$

Step 2.1: Extracting system of recurrences for each inner loop

Inner loop S_1:

$i \in \mathbb{N}$

$$
\begin{cases}
p[i+1] = & p[i]/2 \\
q[i+1] = & q[i]+p[i] \\
r[i+1] = & 2*r[i]-2*q[i]-p[i]
\end{cases}
$$

Inner loop S_2:

$j \in \mathbb{N}$

$$
\begin{cases}
p[j+1] = & p[j]/2 \\
q[j+1] = & q[j] \\
r[j+1] = & 2*r[j],
\end{cases}
$$

where i and j represent the loop counters of the inner loops.

Step 2.2: Solving recurrences for each inner loop

Inner loop S_1:

$i \in \mathbb{N}$

$$
\begin{cases}
p[i] & \underset{C-finite}{=} & \frac{1}{2^i}*p[0_i] \\
q[i] & \underset{Gosper}{=} & q[0_i]+2*p[0_i]-\frac{1}{2^{i-1}}*p[0_i] \\
r[i] & \underset{C-finite}{=} & 2^i*(r[0_i]-2*q[0_i]-2*p[0_i])- \\
 & & \frac{1}{2^{i-1}}*p[0_i]+2*q[0_i]+4*p[0_i]
\end{cases}
$$

Inner loop S_2:

$j \in \mathbb{N}$

$$
\begin{cases}
p[j] & \underset{C-finite}{=} & \frac{1}{2^j}*p[0_j] \\
q[j] & = & q[0_j] \\
r[j] & \underset{C-finite}{=} & 2^j*r[0_j],
\end{cases}
$$

where $p[0_i], q[0_i], r[0_i]$ are the initial values of p,q,r before entering the inner loop S_1. Similarly, $p[0_j], q[0_j], r[0_j]$ are the initial values of p,q,r before entering the inner loop S_2.

Step 2.3: Introducing extra variables with their algebraic dependencies

Inner loop S_1:

$i \in \mathbb{N}$
$x = 2^i, y = 2^{-i}$

$$\begin{cases} p[i] & = & y * p[0_i] \\ q[i] & = & q[0_i] + 2 * p[0_i] - 2 * y * p[0_i] \\ r[i] & = & x * \big(r[0_i] - 2 * q[0_i] - 2 * p[0_i]\big) - \\ & & 2 * y * p[0_i] + 2 * q[0_i] + 4 * p[0_i] \\ 0 & \underset{Alg.Dep}{=} & x * y - 1 \end{cases}$$

Inner loop S_2:

$j \in \mathbb{N}$
$u = 2^j, v = 2^{-j}$

$$\begin{cases} p[j] & = & v * p[0_j] \\ q[j] & = & q[0_j] \\ r[j] & = & u * r[0_j] \\ 0 & \underset{Alg.Dep}{=} & u * v - 1 \end{cases}.$$

Step 2.4: Merged closed form system for $S_1^i; S_2^j$

For the inner loop sequence $S_1^i; S_2^j$, the initial values of the loop variables $p[0_j], q[0_j], r[0_j]$ before entering loop S_2 are given by the values of the variables $p[i], q[i], r[i]$ after S_1^i. Thus, by replacing the closed form expressions of $p[i], q[i], r[i]$ in the closed form system of $p[j], q[j], r[j]$, we get the closed form system for the values of loop variables $p[i, j], q[i, j], r[i, j]$ after $\underbrace{S_1; \ldots; S_1}_{i \text{ times}}; \underbrace{S_2; \ldots; S_2}_{j \text{ times}}$.
Writing p, q, r instead of $p[i, j], q[i, j], r[i, j]$, the obtained merged closed form is as follows.

Inner loop sequence $S_1^i; S_2^j$

$$\begin{cases} p & = & v * y * p[0] \\ q & = & q[0] + 2 * p[0] - 2 * y * p[0] \\ r & = & u * \Big(x * \big(r[0] - 2 * q[0] - 2 * p[0]\big) - 2 * y * p[0] + 2 * q[0] + 4 * p[0]\Big) \\ 0 & = & u * v - 1 \\ 0 & = & x * y - 1, \end{cases}$$

where $p[0], q[0], r[0]$ are the initial values of the loop variables p, q, r before the first iteration of the outer loop (i.e. before $S_1^i; S_2^j$ corresponding to a possible first iteration of the outer loop).

Denote

$$\begin{aligned} I_1 = \; & \langle p - v * y * p[0], q - (q[0] + 2 * p[0] - 2 * y * p[0]), \\ & r - u * \Big(x * \big(r[0] - 2 * q[0] - 2 * p[0]\big) - 2 * y * p[0] + 2 * q[0] + 4 * p[0]\Big) \\ & u * v - 1,\ x * y - 1 \rangle. \end{aligned}$$

Step 2.5: Merged Closed Form System for Loops $S_2^j; S_1^i$

Loop sequence $S_2^j; S_1^i$

$$\begin{cases} p &= y*v*p[0] \\ q &= q[0]+2*p[0]-2*y*p[0] \\ r &= x*(u*r[0]-2*q[0]-2*v*p[0])-2*y*v*p[0]+2*q[0]+4*v*p[0] \\ 0 &= u*v-1 \\ 0 &= x*y-1, \end{cases}$$

where $p[0], q[0], r[0]$ are the initial values of the loop variables p, q, r before the first iteration of the outer loop (i.e. before $S_2^j; S_1^i$ corresponding to a possible first iteration of the outer loop).

Denote

$$\begin{aligned} I_2 = \; & \langle p - v*y*p[0],\ q - q[0] - 2*p[0] + 2*y*p[0] \\ & r - u*\Big(x*\big(r[0] - 2*q[0] - 2*p[0]\big) - 2*y*p[0] + 2*q[0] + 4*p[0]\Big) \\ & u*v-1,\ x*y-1\rangle. \end{aligned}$$

Step 2.6: Elimination of loop counters and new variables

- After eliminating the variables x, y, u, v from I_1 by Gröbner basis computation, the obtained ideal of polynomial relations among the loop variables p, q, r with initial values $p[0], q[0], r[0]$ corresponding to $S_1^i; S_2^j$ is

$$J_1 = \langle q^2 - q[0]^2 + 2*p*r - 2*p[0]*r[0]\rangle.$$

- After eliminating the variables x, y, u, v from I_2 by Gröbner basis computation, the obtained ideal of polynomial relations among the loop variables p, q, r with initial values $p[0], q[0], r[0]$ corresponding to $S_2^j; S_1^i$ is

$$J_2 = \langle q^2 - q[0]^2 + 2*p*r - 2*p[0]*r[0]\rangle.$$

Thus

$$PI_1 = \langle q^2 - q[0]^2 + 2*p*r - 2*p[0]*r[0]\rangle.$$

Step 3: Polynomial invariant for the outer loop

Keeping from (the basis of) PI_1 only those polynomials that are preserved by both S_1 and S_2, we obtain the polynomial invariant

$$GI = \{q^2 - q[0]^2 + 2*p*r - 2*p[0]*r[0]\}.$$

Moreover, by Theorem 6.38, we conclude that *GI* thus obtained generates the ideal of all polynomial invariants for Example 6.42.

Further, in the obtained polynomial invariant we may substitute the concrete values for the symbolically treated initial values $p[0], q[0], r[0]$ given at the entry of the loop. We thus obtain

$$\{a - 2 * p * r - q^2\}.$$

Verification in Theorema

The considered imperative loop is taken from (Zuse, 1993) and implements an algorithm for computing the square root q with precision err for a real number a.

The verification of the algorithm, based on the weakest precondition strategy, is supported in the imperative programming environment of *Theorema* (see Section 3). Thus the specification of the algorithm is as follows.

$$\begin{aligned}&\texttt{Specification}[\text{"}\mathit{SqrtZuse}\text{"},\ \mathit{SqrtZuse}[\downarrow a, \downarrow \mathit{err}, \uparrow q],\\ &\qquad \texttt{Pre} \rightarrow ((a \geq 1) \wedge (\mathit{err} > 0)),\\ &\qquad \texttt{Post} \rightarrow ((q^2 \leq a) \wedge (a < q^2 + \mathit{err}))].\end{aligned}$$

Using the polynomial invariant generated by `Aligator`, as well as other non-polynomial invariant properties, the annotated program code is given below.

$$\begin{aligned}&\texttt{Program}[\text{"}\mathit{SqrtZuse}\text{"},\ \mathit{SqrtZuse}[\downarrow a, \downarrow \mathit{err}, \uparrow q],\\ &\quad \texttt{Module}[\{r, p\},\\ &\qquad r := a - 1;\ q := 1;\ p := 1/2;\\ &\qquad \texttt{WHILE}[2 * p * r \geq \mathit{err},\\ &\qquad\quad \texttt{IF}[2 * r - 2 * q * p \geq 0,\\ &\qquad\qquad r := 2 * r - 2 * q - p; q := q + p; p := p/2,\\ &\qquad\qquad r := 2 * r; p := p/2],\\ &\qquad\quad \texttt{Invariant} \rightarrow (q^2 + 2 * p * r = a) \wedge (\mathit{err} \geq 0) \wedge (p \geq 0) \wedge (r \geq 0) \wedge (q > 0)]]].\end{aligned}$$

Calling the `VCG` for verifying the partial correctness of the "`SqrtZuse`" program w.r.t. its given specification, namely:

$$\texttt{VCG}[\texttt{Program}[\text{"}\mathit{SqrtZuse}\text{"}], \texttt{Specification}[\text{"}\mathit{SqrtZuse}\text{"}]],$$

we obtain 3 universally quantified verification conditions as a *Theorema* lemma, expressed in first order predicate logic.

Lemma (SqrtZuse) :
for any : $\boldsymbol{a}$, $\boldsymbol{err}$, $\boldsymbol{q}$, r, p
(WHILE.Inv)

$$\begin{aligned}&(2 * p * r + \boldsymbol{q}^2 = \boldsymbol{a}) \bigwedge \boldsymbol{err} \geq 0 \bigwedge p \geq 0 \bigwedge r \geq 0 \bigwedge \boldsymbol{q} > 0 \bigwedge 2 * p * r \geq \boldsymbol{err} \Rightarrow\\ &\quad \Big(2 * r + (-2) * p * \boldsymbol{q} \geq 0 \Rightarrow (p * (-1 * p + 2 * r + (-2) * \boldsymbol{q}) + (p + \boldsymbol{q})^2 = \boldsymbol{a}) \bigwedge \boldsymbol{err} \geq 0 \bigwedge\\ &\qquad \frac{1}{2} * p \geq 0 \bigwedge -1 * p + 2 * r + (-2) * \boldsymbol{q} \geq 0 \bigwedge p + \boldsymbol{q} > 0\Big) \bigwedge\\ &\quad \Big(2 * r + (-2) * p * \boldsymbol{q} < 0 \Rightarrow (2 * p * r + \boldsymbol{q}^2 = \boldsymbol{a}) \bigwedge \boldsymbol{err} \geq 0 \bigwedge \frac{1}{2} * p \geq 0 \bigwedge 2 * r \geq 0 \bigwedge \boldsymbol{q} > 0\Big)\end{aligned}$$

(WHILE.Final)
$(2 * p * r + \boldsymbol{q}^2 = \boldsymbol{a}) \wedge \boldsymbol{err} \geq 0 \wedge p \geq 0 \wedge r \geq 0 \wedge \boldsymbol{q} > 0 \wedge 2 * p * r \not\geq \boldsymbol{err} \Rightarrow \boldsymbol{q}^2 \leq \boldsymbol{a} \wedge \boldsymbol{a} < \boldsymbol{err} + \boldsymbol{q}^2$
(Init)
$\boldsymbol{a} \geq 1 \wedge \boldsymbol{err} > 0 \Rightarrow (\boldsymbol{a} = \boldsymbol{a}) \wedge \boldsymbol{err} \geq 0 \wedge -1 + \boldsymbol{a} \geq 0$

6.6 Examples

All results of the examples presented here are outputs of our software package `Aligator` for **a**utomated **l**oop **i**nvariant **g**eneration by **a**lgebraic **t**echniques **o**ver the **r**ationals, implemented in *Mathematica*.

After illustrating our approach for polynomial invariant generation using `Aligator`, for each example of this section we also present how the automatically generated invariants, together with other (user-asserted) non-polynomial invariants, can be further used for imperative program verification in *Theorema*.

The annotation process, i.e. integration of `Aligator` in *Theorema* is not yet computer-supported and is expected to be done manually by the user.

Example 6.43 Given the P-solvable imperative loop with conditional branches and assignments:

$$
\begin{array}{ll}
 & a := x;\ b := y;\ p := 1;\ q := 0;\ r := 0;\ s := 1; \\
 & \texttt{While}[(a \neq b), \\
 & \texttt{If}[a > b \\
S_1: & \quad \texttt{Then}\ a := a - b; p := p - q; r := r - s \\
S_2 & \quad \texttt{Else}\ b := b - a; q := q - p; s := s - r].
\end{array}
$$

By applying Algorithm 6.19 and using `Aligator`, the ideal J_1 of valid polynomial relations among the loop variables a, b, p, q, r, s with initial values $a[0], b[0], p[0], q[0], r[0], s[0]$ for the inner loop sequence $S_1^*; S_2^*$ is given below.

$$
\begin{aligned}
J_1 = \ \langle\ & \\
& -r\ s\ q[0] + s\ q[0] r[0] + q\ r\ s[0] - s\ p[0]\ s[0] - q[0]\ r[0]\ s[0] + p[0]\ s[0]^2, \\
& -r\ q[0] + q[0]\ r[0] + p\ s[0] - p[0]\ s[0], \\
& -q\ r + p\ s + q[0]\ r[0] - p[0]\ s[0], \\
& -s\ b[0]\ p[0] + s\ a[0]\ q[0] + q\ b[0]\ r[0] - b\ q[0]\ r[0] - q\ a[0]\ s[0] + b\ p[0]\ s[0], \\
& -r\ s\ b[0] + s\ b[0]\ r[0] + b\ r\ s[0] - s\ a[0]\ s[0] - b[0]\ r[0]\ s[0] + a[0]\ s[0]^2, \\
& -q\ r\ b[0] + s\ b[0]\ p[0] + b\ r\ q[0] - s\ a[0]\ q[0] - b[0]\ p[0]\ s[0] + a[0]\ q[0]\ s[0], \\
& -p\ q\ b[0] + q\ b[0]\ p[0] + b\ p\ q[0] - q\ a[0]\ q[0] - b[0]\ p[0]\ q[0] + a[0]\ q[0]^2, \\
& -r\ b[0] + b[0]\ r[0] + a\ s[0] - a[0]\ s[0], \\
& -p\ b[0] + b[0]\ p[0] + a\ q[0] - a[0]\ q[0], \\
& -b\ r + a\ s + b[0]\ r[0] - a[0]\ s[0], \\
& -bp + a\ q + b[0]\ p[0] - a[0]\ q[0] \\
\rangle.&
\end{aligned}
$$

Similarly, by applying Algorithm 6.19 and using `Aligator`, the ideal J_2 of polynomial relations among the loop variables a, b, p, q, r, s with initial values $a[0], b[0], p[0], q[0], r[0], s[0]$ for the inner loop sequence $S_2^*; S_1^*$ is given below.

$$
\begin{aligned}
J_2 = \ \langle \\
&-s\ p[0]+q\ r[0]-q[0]\ r[0]+p[0]\ s[0], \\
&-q\ r+p\ s+q[0]\ r[0]-p[0]\ s[0], \\
&-s\ a[0]+b\ r[0]-b[0]\ r[0]+a[0]s[0], \\
&-q\ a[0]+b\ p[0]-b[0]\ p[0]+a[0]\ q[0], \\
&-r\ b[0]\ p[0]+r\ a[0]\ q[0]+p\ b[0]\ r[0]-a\ q[0]\ r[0]-p\ a[0]\ s[0]+a\ p[0]\ s[0], \\
&-b\ r+a\ s+b[0]\ r[0]-a[0]\ s[0], \\
&-bp+a\ q+b[0]\ p[0]-a[0]\ q[0] \\
\rangle.
\end{aligned}
$$

Using Algorithm 6.22, we further compute the intersection ideal of J_1 and J_2 of all valid polynomial relations among the loop variables a,b,p,q,r,s with initial values $a[0],b[0],p[0],q[0],r[0],s[0]$ corresponding to the first iteration of the outer loop. A basis for $PI_1 = J_1 \cap J_2$ is hence computed by `Aligator`, and it is formed by 18 polynomial relations (see Example 7.26 for the concrete set of polynomials).

Finally, by applying step 3 of Algorithm 6.32 on PI_1, the set GI of polynomial invariants returned by `Aligator` is as follows.

$$
\begin{aligned}
GI = \ \{ \\
&-b\ p+a\ q+b[0]\ p[0]-a[0]\ q[0], \\
&-b\ r+a\ s+b[0]\ r[0]-a[0]\ s[0], \\
&-q\ r+p\ s+q[0]\ r[0]-p[0]\ s[0], \\
&-r\ b[0]\ p[0]+r\ a[0]\ q[0]+p\ b[0]\ r[0]-a\ q[0]\ r[0]-p\ a[0]\ s[0]+a\ p[0]\ s[0], \\
&-s\ b[0]\ p[0]+s\ a[0]\ q[0]+q\ b[0]\ r[0]-b\ q[0]\ r[0]-q\ a[0]\ s[0]+b\ p[0]\ s[0] \\
\}.
\end{aligned}
$$

Moreover, by Theorem 6.37, we conclude that the polynomials thus obtained generate the ideal of all polynomial invariants for Example 6.43.

Further, we may substitute the concrete values for the symbolically treated initial values $a[0],b[0]$, $p[0],q[0],r[0],s[0]$ given at the entry of the loop. From GI we thus obtain

$$\{\ -bp+aq+y,\ -br+as-x,\ -1-qr+ps,\ a-px-ry,\ b-qx-sy\ \}.$$

Verification in Theorema

The considered imperative loop is taken from (Knuth, 1998; Rodriguez-Carbonell and Kapur, 2004) and implements the extended Euclid's algorithm, for computing the greatest common divisor G of two integer numbers x, y, together with the Bézout's coefficients (p,r), i.e. $a = p*x+r*y$.

The verification of the extended Euclid's algorithm, based on the weakest precondition strategy, is supported in the imperative programming environment of *Theorema* (see Section 3). Thus the

specification of the algorithm is as follows.

$$\begin{aligned}&\texttt{Specification}[\text{"}\textit{GCDextended}\text{"},\ \textit{GCDExt}[\downarrow x, \downarrow y, \uparrow G],\\ &\qquad \texttt{Pre} \rightarrow ((x>0)\wedge(y>0)),\\ &\qquad \texttt{Post} \rightarrow (G = \texttt{GCD}[x,y])].\end{aligned}$$

Using the polynomial invariants generated by `Aligator` and other non-polynomial invariant properties, the annotated program code is given below.

$$\begin{aligned}&\texttt{Program}[\text{"}\textit{GCDextended}\text{"},\ \textit{GCDExt}[\downarrow x, \downarrow y, \uparrow G],\\ &\quad \texttt{Module}[\{a,b,p,q,r,s\},\\ &\qquad a := x; b := y; p := 1; q := 0; r := 0; s := 1;\\ &\qquad \texttt{WHILE}[a \neq b,\\ &\qquad\quad \texttt{IF}[a > b,\\ &\qquad\qquad a := a-b; p := p-q; r := r-s,\\ &\qquad\qquad b := b-a; q := q-p; s := s-r],\\ &\qquad\quad \texttt{Invariant} \rightarrow (p*s-q*r=1)\wedge(b=q*x+s*y)\wedge(a=p*x+r*y)\wedge\\ &\qquad\qquad\qquad\qquad (x=a*s-b*r)\wedge(y=b*p-a*q)\wedge\\ &\qquad\qquad\qquad\qquad (\texttt{GCD}[a,b]=\texttt{GCD}[x,y])\wedge(a>0)\wedge(b>0)];\\ &\quad G := a]].\end{aligned}$$

Calling the `VCG` for verifying the partial correctness of the "`GCDextended`" program w.r.t. its given specification, namely:

$$\texttt{VCG}[\texttt{Program}[\text{"}\textit{GCDextended}\text{"}], \texttt{Specification}[\text{"}\textit{GCDextended}\text{"}]],$$

we obtain 3 universally quantified verification conditions as a *Theorema* lemma, expressed in first order predicate logic.

Lemma (GCDextended) :
for any : x, y, G, a, b, p, q, r, s
(WHILE.Inv)
$(-1*q*r+p*s=1)\wedge(b=q*x+s*y)\wedge(a=p*x+r*y)\wedge(x=-1*b*r+a*s)\wedge$
$(y=b*p+(-1)*a*q)\wedge(\mathrm{GCD}[a,b]=\mathrm{GCD}[x,y])\wedge a>0\wedge b>0\wedge a\neq b\Rightarrow$
$(a>b\Rightarrow(-1*q*(r+(-1)*s)+(p+(-1)*q)*s=1)\wedge(b=q*x+s*y)\wedge$
$(a+(-1)*b=(p+(-1)*q)*x+(r+(-1)*s)*y)\wedge(x=-1*b*(r+(-1)*s)+(a+(-1)*b)*s)\wedge$
$(y=b*(p+(-1)*q)+(-1)*(a+(-1)*b)*q)\wedge(\mathrm{GCD}[b,a+(-1)*b]=\mathrm{GCD}[x,y])\wedge$
$a+(-1)*b>0\wedge b>0)\wedge$
$(a\leq b\Rightarrow(-1*(-1*p+q)*r+p*(-1*r+s)=1)\wedge(-1*a+b=(-1*p+q)*x+(-1*r+s)*y)\wedge$
$(a=p*x+r*y)\wedge(x=-1*(-1*a+b)*r+a*(-1*r+s))\wedge$
$(y=(-1*a+b)*p+(-1)*a*(-1*p+q))\wedge(\mathrm{GCD}[a,-1*a+b]=\mathrm{GCD}[x,y])\wedge$
$a>0\wedge-1*a+b>0)$
(WHILE.Final)
$(-1*q*r+p*s=1)\wedge(b=q*x+s*y)\wedge(a=p*x+r*y)\wedge(x=-1*b*r+a*s)\wedge$
$(y=b*p+(-1)*a*q)\wedge(\mathrm{GCD}[a,b]=\mathrm{GCD}[x,y])\wedge a>0\wedge b>0\wedge\neg(a\neq b)\Rightarrow(a=\mathrm{GCD}[x,y])$
(Init)
$x>0\wedge y>0\Rightarrow(1=1)\wedge(y=y)\wedge(x=x)\wedge(x=x)\wedge(y=y)\wedge(\mathrm{GCD}[x,y]=\mathrm{GCD}[x,y])\wedge x>0\wedge y>0$

Example 6.44 Given the P-solvable imperative loop with conditional branches and assignments:

$$
\begin{aligned}
&x:=a;y:=b;z:=0;\\
&\texttt{While}[(y\neq 0),\\
&\texttt{If}[\texttt{Mod}[y,2]=1\\
&\ \ \texttt{Then}\ z:=z+x;y:=y-1];\\
&x:=2*x;y:=y/2].
\end{aligned}
$$

By applying Algorithm 6.32 and using `Aligator`, we obtain the polynomial invariant

$$\{\, x\,y+z-x[0]\,y[0]-z[0]\,\}.$$

Moreover, by Theorem 6.38, we conclude that the polynomial thus obtained generates the ideal of all polynomial invariants for Example 6.44.

Further, in the obtained polynomial invariant we may substitute the concrete values for the symbolically treated initial values $x[0], y[0], z[0]$ given at the entry of the loop. We thus obtain

$$\{-ab+xy+z\}.$$

Verification in Theorema

The considered imperative loop is taken from (Knuth, 1998) and implements an algorithm for computing the product of two natural numbers a, b.

In the imperative verification environment of *Theorema* (see Section 3), the specification of the algorithm is as follows.

$$\begin{aligned}&\texttt{Specification}[\text{"}\textit{Product}\text{"},\ \textit{Prod}[\downarrow a, \downarrow b, \uparrow z],\\ &\qquad \texttt{Pre} \rightarrow ((a \geq 0) \wedge (b \geq 0)),\\ &\qquad \texttt{Post} \rightarrow (z = a * b)].\end{aligned}$$

Using the polynomial invariant generated by `Aligator`, as well as other non-polynomial invariant properties, the annotated program code is given below.

$$\begin{aligned}&\texttt{Program}[\text{"}\textit{Product}\text{"},\ \textit{Prod}[\downarrow a, \downarrow b, \uparrow z],\\ &\quad \texttt{Module}[\{x, y\},\\ &\qquad x := a; y := b; z := 0\\ &\qquad \texttt{WHILE}[y \neq 0,\\ &\qquad\quad \texttt{IF}[\texttt{Mod}[y, 2] = 1,\\ &\qquad\qquad z := z + x; y := y - 1];\\ &\qquad\quad x := 2 * x; y := y/2,\\ &\qquad\quad \texttt{Invariant} \rightarrow (z + x * y = a * b) \wedge (x \geq 0) \wedge (y \geq 0) \wedge (z \geq 0)]]].\end{aligned}$$

Calling the `VCG` for verifying the partial correctness of the "`Product`" program w.r.t. its given specification, namely:

$$\texttt{VCG}[\texttt{Program}[\text{"}\textit{Product}\text{"}], \texttt{Specification}[\text{"}\textit{Product}\text{"}]],$$

we obtain 3 universally quantified verification conditions as a *Theorema* lemma, expressed in first order predicate logic.

Lemma (Product) :
for any : $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{z}$, x, y

(WHILE.Inv)

$$(x * y + \boldsymbol{z} = \boldsymbol{a} * \boldsymbol{b}) \wedge x \geq 0 \wedge y \geq 0 \wedge \boldsymbol{z} \geq 0 \wedge y \neq 0 \Rightarrow$$
$$\left((\text{Mod}[y,\ 2] = 1) \Rightarrow (x + x * (-1 + y) + \boldsymbol{z} = \boldsymbol{a} * \boldsymbol{b}) \bigwedge 2 * x \geq 0 \bigwedge \frac{1}{2} * (-1 + y) \geq 0 \bigwedge x + \boldsymbol{z} \geq 0\right) \bigwedge$$
$$\left(\neg\ (\text{Mod}[y,\ 2] = 1) \Rightarrow (x * y + \boldsymbol{z} = \boldsymbol{a} * \boldsymbol{b}) \bigwedge 2 * x \geq 0 \bigwedge \frac{1}{2} * y \geq 0 \bigwedge \boldsymbol{z} \geq 0\right)$$

(WHILE.Final)

$$(x * y + \boldsymbol{z} = \boldsymbol{a} * \boldsymbol{b}) \wedge x \geq 0 \wedge y \geq 0 \wedge \boldsymbol{z} \geq 0 \wedge \neg\ (y \neq 0) \Rightarrow (\boldsymbol{z} = \boldsymbol{a} * \boldsymbol{b})$$

(Init)

$$\boldsymbol{a} \geq 0 \wedge \boldsymbol{b} \geq 0 \Rightarrow (\boldsymbol{a} * \boldsymbol{b} = \boldsymbol{a} * \boldsymbol{b}) \wedge \boldsymbol{a} \geq 0 \wedge \boldsymbol{b} \geq 0$$

Example 6.45 Given the P-solvable imperative loop with conditional branches and assignments:

$$\begin{aligned}&q := 0; r := A;\\ &\texttt{While}[(b \neq B),\\ &q := 2 * q;\ b := b/2;\\ &\texttt{If}[(r \geq b)\\ &\ \texttt{Then}\ q := q + 1; r := r - b]].\end{aligned}$$

By applying Algorithm 6.32 and using `Aligator`, we obtain the polynomial invariant

$$\{\ -b\,q + b[0]\,q[0] - r + r[0]\ \}.$$

Moreover, by Theorem 6.38, we conclude that the polynomial thus obtained generates the ideal of all polynomial invariants for Example 6.45.

Further, in the obtained polynomial invariant we may substitute the concrete values for the symbolically treated initial values $b[0], q[0], r[0]$ given at the entry of the loop. We thus obtain

$$\{\ A - bq - r\ \}.$$

Verification in Theorema

The considered imperative loop is taken from (Kaldewaij, 1990) and implements a part of an algorithm for computing the quotient q and remainder r of two integer numbers $A,\ B$.

In the imperative verification environment of *Theorema* (see Section 3), the specification of the algorithm is as follows.

$$\begin{array}{l}\texttt{Specification}[\text{"}\textit{DivKald}\text{"},\ \textit{DivKald}[\downarrow A, \downarrow B, \uparrow r, \uparrow q],\\ \quad \texttt{Pre} \rightarrow ((A \geq 0) \wedge (B > 0)),\\ \quad \texttt{Post} \rightarrow ((r = \texttt{Mod}[A,B]) \wedge (q = \texttt{Quotient}[A,B]))].\end{array}$$

Using the polynomial invariant generated by `Aligator`, as well as other non-polynomial invariant properties, the annotated program code with two loops is given below.

$$\begin{array}{l}\texttt{Program}[\text{"}\textit{DivKald}\text{"},\ \textit{DivKald}[\downarrow A, \downarrow B, \uparrow r, \uparrow q],\\ \quad \texttt{Module}[\{b\},\\ \quad\quad q := 0; r := A; b := B;\\ \quad\quad \texttt{WHILE}[r \geq b,\\ \quad\quad\quad b := 2 * b,\\ \quad\quad\quad \texttt{Invariant} \rightarrow \left(\underset{k}{\exists}\ \left((k \geq 0) \wedge (b = 2^k * B)\right)\right)\\ \quad\quad\quad\quad\quad\quad (q = 0) \wedge (r = A) \wedge (A \geq 0) \wedge (B > 0)];\\ \quad\quad \texttt{WHILE}[b \neq B,\\ \quad\quad\quad q := 2 * q; b := b/2;\\ \quad\quad\quad \texttt{IF}[r \geq b,\\ \quad\quad\quad\quad q := q + 1; r := r - b],\\ \quad\quad\quad \texttt{Invariant} \rightarrow (A = q * b + r) \wedge (r \geq 0) \wedge\\ \quad\quad\quad \left(\underset{k}{\exists}\ \left((k \geq 0) \wedge (b = 2^k * B)\right)\right) \wedge (B > 0) \wedge (b > r) \wedge (q \geq 0)]]].\end{array}$$

The generated invariant by `Aligator` is used in annotating the second loop of the above program.

Calling the `VCG` for verifying the partial correctness of the "`DivKald`" program w.r.t. its given specification, namely:

$$\texttt{VCG}[\texttt{Program}[\text{“}\textit{DivKald}\text{”}], \texttt{Specification}[\text{“}\textit{DivKald}\text{”}]],$$

we obtain 5 universally quantified verification conditions as a *Theorema* lemma, expressed in first order predicate logic.

Lemma (DivKald) :

for any : A, B, r, q, b

(WHILE.Inv)

$$(A = b*q + r) \bigwedge r \geq 0 \bigwedge \exists_k (k \geq 0 \wedge (b = 2^k * B)) \bigwedge B > 0 \bigwedge b > r \bigwedge q \geq 0 \bigwedge b \neq B \Rightarrow$$
$$\left(r \geq \frac{1}{2} * b \Rightarrow \left(A = \frac{-1}{2} * b + \frac{1}{2} * b * (1 + 2 * q) + r\right) \bigwedge \frac{-1}{2} * b + r \geq 0 \bigwedge \exists_k \left(k \geq 0 \bigwedge \left(\frac{1}{2} * b = 2^k * B\right)\right) \bigwedge \right.$$
$$\left. B > 0 \bigwedge \frac{1}{2} * b > \frac{-1}{2} * b + r \bigwedge 1 + 2 * q \geq 0\right) \bigwedge$$
$$\left(r < \frac{1}{2} * b \Rightarrow (A = b*q + r) \bigwedge r \geq 0 \bigwedge \exists_k \left(k \geq 0 \bigwedge \left(\frac{1}{2} * b = 2^k * B\right)\right) \bigwedge B > 0 \bigwedge \frac{1}{2} * b > r \bigwedge 2 * q \geq 0\right)$$

(WHILE.Final)

$$(A = b*q + r) \bigwedge r \geq 0 \bigwedge \exists_k (k \geq 0 \wedge (b = 2^k * B)) \bigwedge B > 0 \bigwedge b > r \bigwedge q \geq 0 \bigwedge \neg (b \neq B) \Rightarrow$$
$$(r = \text{Mod}[A, B]) \wedge (q = \text{Quotient}[A, B])$$

(WHILE.Inv)

$$\exists_k (k \geq 0 \wedge (b = 2^k * B)) \bigwedge (q = 0) \bigwedge (r = A) \bigwedge A \geq 0 \bigwedge B > 0 \bigwedge r \geq b \Rightarrow$$
$$\exists_k (k \geq 0 \wedge (2 * b = 2^k * B)) \bigwedge (q = 0) \bigwedge (r = A) \bigwedge A \geq 0 \bigwedge B > 0$$

(WHILE.Final)

$$\exists_k (k \geq 0 \wedge (b = 2^k * B)) \bigwedge (q = 0) \bigwedge (r = A) \bigwedge A \geq 0 \bigwedge B > 0 \bigwedge r \ngeq b \Rightarrow$$
$$(A = b*q + r) \bigwedge r \geq 0 \bigwedge \exists_k (k \geq 0 \wedge (b = 2^k * B)) \bigwedge B > 0 \bigwedge b > r \bigwedge q \geq 0$$

(Init)

$$A \geq 0 \wedge B > 0 \Rightarrow \exists_k (k \geq 0 \wedge (B = 2^k * B)) \bigwedge (0 = 0) \bigwedge (A = A) \bigwedge A \geq 0 \bigwedge B > 0$$

Example 6.46 Given the P-solvable imperative loop with conditional branches and assignments:

$$r := R^2 - N; u := 2 * R + 1; v := 1;$$
$$\texttt{While}[(r \neq 0),$$
$$\texttt{If}[(r > 0)$$
$$\texttt{Then}\ r := r - v; v := v + 2$$
$$\texttt{Else}\ r := r + u; u := u + 2]].$$

By applying Algorithm 6.32 and using `Aligator`, we obtain the polynomial invariant

$$\{\ 4\ r-4\ r[0]+2\ u-u^2-2\ u[0]+u[0]^2-2\ v+v^2+2\ v[0]-v[0]^2\ \}.$$

Moreover, by Theorem 6.38, we conclude that the polynomial thus obtained generates the ideal of all polynomial invariants for Example 6.46.

Further, in the obtained polynomial invariant we may substitute the concrete values for the symbolically treated initial values $r[0], u[0], v[0]$ given at the entry of the loop. We thus obtain

$$\{\ 4N+4r+2u-u^2-2v+v^2\ \}.$$

Verification in Theorema

The considered imperative loop is taken from (Knuth, 1998) and implements Fermat's algorithm for finding integer factors $a,\ b$ for a given integer number $N \geq 1$ only by addition and subtraction. The algorithm starts from the square root R of N, and works outwards. The program finds two numbers $x,\ y$ such that N can be written as a difference of the squares of these numbers, i.e. $N = x^2 - y^2$, and thus N is represented as the product of two integers $a,\ b$.

In the imperative verification environment of *Theorema* (see Section 3), the specification of the algorithm is as follows.

$$\begin{aligned}&\texttt{Specification}[\text{"}\textit{Fermat}\text{"},\ \textit{Ferm}[\downarrow N, \downarrow R, \uparrow x, \uparrow y, \uparrow a, \uparrow b],\\ &\quad \texttt{Pre} \rightarrow ((N \geq 1) \wedge ((R-1)^2 < N \leq R^2) \wedge (\texttt{Mod}[N,2] = 1)),\\ &\quad \texttt{Post} \rightarrow ((N = x^2 - y^2) \wedge (N = a * b))].\end{aligned}$$

Using the polynomial invariant generated by `Aligator`, as well as other non-polynomial invariant properties, the annotated program code is given below.

$$\begin{aligned}&\texttt{Program}[\text{"}\textit{Fermat}\text{"},\ \textit{Ferm}[\downarrow N, \downarrow R, \uparrow x, \uparrow y, \uparrow a, \uparrow b],\\ &\quad \texttt{Module}[\{u, v, r\},\\ &\qquad u := 2 * R + 1; v := 1; r := R * R - N;\\ &\qquad \texttt{WHILE}[r \neq 0,\\ &\qquad\quad \texttt{IF}[r > 0,\\ &\qquad\qquad r := r - v; v := v + 2,\\ &\qquad\qquad r := r + u; u := u + 2],\\ &\qquad\quad \texttt{Invariant} \rightarrow (4 * (N + r) = u^2 - v^2 + 2 * v - 2 * u) \wedge (u + v - 2 \leq 2 * N) \wedge\\ &\qquad\qquad\qquad (\texttt{Mod}[u,2] = 1) \wedge (\texttt{Mod}[v,2] = 1)];\\ &\quad a := (u - v)/2; b := (u + v - 2)/2; x := (a + b)/2; y := (b - a)/2]].\end{aligned}$$

Calling the `VCG` for verifying the partial correctness of the "`Fermat`" program w.r.t. its given specification, namely:

$$\texttt{VCG}[\texttt{Program}[\text{"}\textit{Fermat}\text{"}], \texttt{Specification}[\text{"}\textit{Fermat}\text{"}]],$$

we obtain 3 universally quantified verification conditions as a *Theorema* lemma, expressed in first order predicate logic.

Lemma (Fermat) :

for any : N, R, x, y, a, b, u, v, r

(WHILE.Inv)

$$(4*(r+N) = -2*u+u^2+2*v+(-1)*v^2) \bigwedge -2+u+v \le 2*N \bigwedge (\text{Mod}[u,\,2] = 1) \bigwedge (\text{Mod}[v,\,2] = 1) \bigwedge r \neq 0 \Rightarrow$$
$$(r > 0 \Rightarrow (4*(r+(-1)*v+N) = -2*u+u^2+2*(2+v)+(-1)*(2+v)^2) \bigwedge u+v \le 2*N \bigwedge$$
$$(\text{Mod}[u,\,2] = 1) \bigwedge (\text{Mod}[2+v,\,2] = 1)) \bigwedge$$
$$(r \le 0 \Rightarrow (4*(r+u+N) = -2*(2+u)+(2+u)^2+2*v+(-1)*v^2) \bigwedge u+v \le 2*N \bigwedge$$
$$(\text{Mod}[2+u,\,2] = 1) \bigwedge (\text{Mod}[v,\,2] = 1))$$

(WHILE.Final)

$$(4*(r+N) = -2*u+u^2+2*v+(-1)*v^2) \bigwedge -2+u+v \le 2*N \bigwedge (\text{Mod}[u,\,2] = 1) \bigwedge (\text{Mod}[v,\,2] = 1) \bigwedge \neg (r \neq 0) \Rightarrow$$
$$\left(N = \frac{1}{4}*\left(\frac{1}{2}*(u+(-1)*v)+\frac{1}{2}*(-2+u+v)\right)^2+\left(\frac{-1}{4}\right)*\left(\frac{1}{2}*(-1*u+v)+\frac{1}{2}*(-2+u+v)\right)^2\right) \bigwedge$$
$$\left(N = \frac{1}{4}*(u+(-1)*v)*(-2+u+v)\right)$$

(Init)

$$N \ge 1 \bigwedge (-1+R)^2 < N \bigwedge N \le R^2 \bigwedge (\text{Mod}[N,\,2] = 1) \Rightarrow$$
$$(4*R^2 = 1+(-2)*(1+2*R)+(1+2*R)^2) \bigwedge 2*R \le 2*N \bigwedge (\text{Mod}[1+2*R,\,2] = 1) \bigwedge (1 = 1)$$

Example 6.47 Given the P-solvable imperative loop with conditional branches and assignments:

$$\begin{aligned}
&a := 0; b := Q/2; d := 1; y := 0;\\
&\texttt{While}[(d \ge Tol),\\
&\texttt{If}[(P < a+b)\\
&\ \texttt{Then}\ b := b/2; d := d/2\\
&\ \texttt{Else}\ a := a+b; y := y+d/2; b := b/2; d := d/2]].
\end{aligned}$$

As presented already in Example 6.39, the obtained polynomial invariants

$$\begin{aligned}
&\{\ -b[0]\ d + b\ d[0],\\
&\ a\ d - a[0]\ d - 2\ b\ y + 2\ b\ y[0],\\
&\ a\ d[0] - a[0]\ d[0] - 2\ b[0]\ y + 2\ b[0]\ y[0]\ \}
\end{aligned}$$

represent a basis of the polynomial invariant ideal Example 6.47.

Further, in the obtained polynomial invariants we may substitute the concrete values for the symbolically treated initial values $a[0], b[0], d[0], y[0]$ given at the entry of the loop. We thus obtain

$$\{\ b - \frac{dQ}{2},\ ad - 2by,\ a - Qy\ \}.$$

Verification in Theorema

The considered imperative loop is taken from (Wegbreit, 1974) and implements Wensley's algorithm for real division. The algorithm computes the quotient r of the division of two real numbers P and Q, with a precision Tol.

In the imperative verification environment of *Theorema* (see Section 3), the specification of the algorithm is as follows.

$$\begin{array}{l}\texttt{Specification}[\text{"}\textit{Wensley}\text{"},\ \textit{Wensley}[\downarrow P, \downarrow Q, \downarrow Tol, \uparrow r] \\ \quad \texttt{Pre} \rightarrow ((Q > P \geq 0) \wedge (Tol > 0), \\ \quad \texttt{Post} \rightarrow ((P/Q < r + Tol) \wedge (r \leq P/Q))].\end{array}$$

Using the polynomial invariant generated by `Aligator`, as well as other non-polynomial invariant properties, the annotated program code is given below.

$$\begin{array}{l}\texttt{Program}[\text{"}\textit{Wensley}\text{"},\ \textit{Wensley}[\downarrow P, \downarrow Q, \downarrow Tol, \uparrow r] \\ \quad \texttt{Module}[\{a,b,d,y\}, \\ \quad\quad u := 2*R+1; v := 1; r := R*R-N; \\ \quad\quad \texttt{WHILE}[d \geq Tol, \\ \quad\quad\quad \texttt{IF}[P < a+b, \\ \quad\quad\quad\quad b := b/2; d := d/2, \\ \quad\quad\quad\quad a := a+b; y := y+d/2; b := b/2; d := d/2], \\ \quad\quad\quad \texttt{Invariant} \rightarrow (a = Q*y) \wedge (b = d*Q/2) \wedge (a*d - 2*b*y) \wedge \\ \quad\quad\quad\quad\quad\quad (y \leq P/Q < y+d) \wedge (0 < d \leq 1) \wedge (Q > P \geq 0)]; \\ \quad r := y]].\end{array}$$

Calling the `VCG` for verifying the partial correctness of the "`Wensley`" program w.r.t. its given specification, namely:

$$\texttt{VCG}[\texttt{Program}[\text{"}\textit{Wensley}\text{"}], \texttt{Specification}[\text{"}\textit{Wensley}\text{"}]],$$

we obtain 3 universally quantified verification conditions as a *Theorema* lemma, expressed in first order predicate logic.

Lemma (Wensley) :
for any : P, Q, Tol, r, a, b, d, y
(WHILE.Inv)

$$(a = y*Q) \bigwedge \left(b = \frac{1}{2}*d*Q\right) \bigwedge (a*d = 2*b*y) \bigwedge$$
$$y \le P*Q^{-1} \bigwedge P*Q^{-1} < d + y \bigwedge 0 < d \bigwedge d \le 1 \bigwedge Q > P \bigwedge P \ge 0 \bigwedge d \ge Tol \Rightarrow$$
$$\left(P < a + b \Rightarrow (a = y*Q) \bigwedge \left(\frac{1}{2}*b = \frac{1}{4}*d*Q\right) \bigwedge \left(\frac{1}{2}*a*d = b*y\right) \bigwedge y \le P*Q^{-1} \bigwedge P*Q^{-1} < \frac{1}{2}*d + y \bigwedge\right.$$
$$\left. 0 < \frac{1}{2}*d \bigwedge \frac{1}{2}*d \le 1 \bigwedge Q > P \bigwedge P \ge 0\right) \bigwedge$$
$$\left(P \ge a + b \Rightarrow \left(a + b = \left(\frac{1}{2}*d + y\right)*Q\right) \bigwedge \left(\frac{1}{2}*b = \frac{1}{4}*d*Q\right) \bigwedge \left(\frac{1}{2}*(a+b)*d = b*\left(\frac{1}{2}*d + y\right)\right) \bigwedge\right.$$
$$\left. \frac{1}{2}*d + y \le P*Q^{-1} \bigwedge P*Q^{-1} < d + y \bigwedge 0 < \frac{1}{2}*d \bigwedge \frac{1}{2}*d \le 1 \bigwedge Q > P \bigwedge P \ge 0\right)$$

(WHILE.Final)

$$(a = y*Q) \bigwedge \left(b = \frac{1}{2}*d*Q\right) \bigwedge (a*d = 2*b*y) \bigwedge y \le P*Q^{-1} \bigwedge$$
$$P*Q^{-1} < d + y \bigwedge 0 < d \bigwedge d \le 1 \bigwedge Q > P \bigwedge P \ge 0 \bigwedge d \not\ge Tol \Rightarrow P*Q^{-1} < y + Tol \wedge y \le P*Q^{-1}$$

(Init)

$$Q > P \wedge P \ge 0 \wedge Tol > 0 \Rightarrow$$
$$(0 = 0) \bigwedge \left(\frac{1}{2}*Q = \frac{1}{2}*Q\right) \bigwedge (0 = 0) \bigwedge 0 \le P*Q^{-1} \bigwedge P*Q^{-1} < 1 \bigwedge Q > P \bigwedge P \ge 0$$

Example 6.48 Given the imperative loop with conditional branches and affine assignments:

$$\begin{array}{l} x := a;\ y := b;\ u := b;\ v := a; \\ \texttt{While}[x \neq y, \\ \quad \texttt{If}[x > y \\ \quad\quad \texttt{Then}\ x := x - y; v := v + u, \\ \quad\quad \texttt{Else}\ y := y - x; u := u + v]]. \end{array}$$

By applying Algorithm 6.32 and using `Aligator`, the obtained polynomial invariant is

$$\{\ u\,x - u[0]\,x[0] + v\,y - v[0]\,y[0]\ \}.$$

Moreover, by Theorem 6.37, we conclude that the polynomial thus obtained generates the ideal of all polynomial invariants for Example 6.48.

Further, in the obtained polynomial invariant we may substitute the concrete values for the symbolically treated initial values $u[0], v[0], x[0], y[0]$ given at the entry of the loop. We thus obtain

$$\{\ -2ab + ux + vy\ \}.$$

Verification in Theorema

The considered imperative loop is taken from (Dijkstra, 1976; Sankaranaryanan *et al.*, 2004) and implements an algorithm for simultaneously computing the least common multiple L and the greatest common divisor G of two integers a and b.

In the imperative verification environment of *Theorema* (see Section 3), the specification of the algorithm is as follows.

$$\begin{aligned}&\texttt{Specification}[\text{"}\mathit{LCM-GCD}\text{"},\ \mathit{LcmGcd}[\downarrow a, \downarrow b, \uparrow G, \uparrow L],\\ &\qquad \texttt{Pre} \rightarrow ((a>0)\wedge(b>0)),\\ &\qquad \texttt{Post} \rightarrow ((L=\texttt{LCM}[a,b])\wedge(G=\texttt{GCD}[a,b]))].\end{aligned}$$

Using the polynomial invariant generated by `Aligator` and other non-polynomial invariant properties, the annotated program code is given below.

$$\begin{aligned}&\texttt{Program}[\text{"}\mathit{LCM-GCD}\text{"},\ \mathit{LcmGcd}[\downarrow a, \downarrow b, \uparrow G, \uparrow L],\\ &\quad \texttt{Module}[\{x,y,u,v\},\\ &\qquad x:=a;\ y:=b;\ u:=b;\ v:=a;\\ &\qquad \texttt{WHILE}[(x\neq y),\\ &\qquad\quad \texttt{IF}[x>y,\\ &\qquad\qquad x:=x-y;\ v:=v+u,\\ &\qquad\qquad y:=y-x;\ u:=u+v],\\ &\qquad \texttt{Invariant} \rightarrow (u*x+v*y=2*a*b)\wedge(\texttt{GCD}[x,y]=\texttt{GCD}[a,b])\wedge(x>0)\wedge(y>0)];\\ &\qquad L:=(u+v)/2;\ G:=x]].\end{aligned}$$

Calling the `VCG` for verifying the partial correctness of the "`LCM-GCD`" program w.r.t. its given specification, namely:

$$\texttt{VCG}[\texttt{Program}[\text{"}\mathit{LCM-GCD}\text{"}], \texttt{Specification}[\text{"}\mathit{LCM-GCD}\text{"}]],$$

we obtain 3 universally quantified verification conditions as a *Theorema* lemma, expressed in first order predicate logic.

Lemma (LCM – GCD) :

for any : $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{G}$, $\boldsymbol{L}$, x, y, u, v

(WHILE.Inv)

$$\begin{aligned}&(u*x+v*y=2*\boldsymbol{a}*\boldsymbol{b})\wedge(\text{GCD}[x,\ y]=\text{GCD}[\boldsymbol{a},\ \boldsymbol{b}])\wedge x>0\wedge y>0\wedge x\neq y\Rightarrow\\ &(x>y\Rightarrow(u*(x+(-1)*y)+(u+v)*y=2*\boldsymbol{a}*\boldsymbol{b})\wedge(\text{GCD}[y,\ x+(-1)*y]=\text{GCD}[\boldsymbol{a},\ \boldsymbol{b}])\wedge\\ &\qquad x+(-1)*y>0\wedge y>0)\wedge\\ &(x\leq y\Rightarrow((u+v)*x+v*(-1*x+y)=2*\boldsymbol{a}*\boldsymbol{b})\wedge(\text{GCD}[x,\ -1*x+y]=\text{GCD}[\boldsymbol{a},\ \boldsymbol{b}])\wedge\\ &\qquad x>0\wedge-1*x+y>0)\end{aligned}$$

(WHILE.Final)

$$\begin{aligned}&(u*x+v*y=2*\boldsymbol{a}*\boldsymbol{b})\wedge(\text{GCD}[x,\ y]=\text{GCD}[\boldsymbol{a},\ \boldsymbol{b}])\wedge x>0\wedge y>0\wedge\neg(x\neq y)\Rightarrow\\ &\left(\frac{1}{2}*(u+v)=\text{LCM}[\boldsymbol{a},\ \boldsymbol{b}]\right)\bigwedge(x=\text{GCD}[\boldsymbol{a},\ \boldsymbol{b}])\end{aligned}$$

(Init)

$$\boldsymbol{a}>0\wedge\boldsymbol{b}>0\Rightarrow(2*\boldsymbol{a}*\boldsymbol{b}=2*\boldsymbol{a}*\boldsymbol{b})\wedge(\text{GCD}[\boldsymbol{a},\ \boldsymbol{b}]=\text{GCD}[\boldsymbol{a},\ \boldsymbol{b}])\wedge\boldsymbol{a}>0\wedge\boldsymbol{b}>0$$

Example 6.49 The following example illustrates the applicability of our approach in the case of P-solvable loops with more than 2 conditional branches.

Given the P-solvable imperative loop with conditional branches and assignments:

$$
\begin{aligned}
&s := \texttt{Sqrt}[n];\\
&d := f; r := \texttt{Mod}[n,d]; rp := \texttt{Mod}[n,d-2];\\
&q := 4*(\texttt{Quotient}[n,d-2] - \texttt{Quotient}[n,d]);\\
&\texttt{While}[((s \geq d) \wedge (r \neq 0)),\\
&\texttt{If}[(2*r - rp + q < 0)\\
&\ \texttt{Then}\ t := r; r := 2*r - rp + q + d + 2; rp := t; q := q + 4; d := d + 2\\
&\ \texttt{Else If}[((2*r - rp + q \geq 0) \wedge (2*r - rp + q < d + 2))\\
&\ \texttt{Then}\ t := r; r := 2*r - rp + q; rp := t; d := d + 2\\
&\ \texttt{Else If}[((2*r - rp + q \geq 0) \wedge (2*r - rp + q \geq d + 2) \wedge (2*r - rp + q < 2*d + 4))\\
&\ \texttt{Then}\ t := r; r := 2*r - rp + q - d - 2; rp := t; q := q - 4; d := d + 2\\
&\ \texttt{Else}\ t := r; r := 2*r - rp + q - 2*d - 4; rp := t; q := q - 8; d := d + 2]]]].
\end{aligned}
$$

By applying Algorithm 6.32 and using `Aligator`, the obtained polynomial invariant is

$$\{2dq - d^2q - 8r + 4dr - 4d\ rp - 2\ d[0]\ q[0] + d[0]^2\ q[0] + 8\ r[0] - 4\ d[0]\ r[0] + 4\ d[0]\ rp[0]\}.$$

Moreover, by the third condition of Theorem 6.40, we conclude that the polynomial thus obtained generates the polynomial invariant ideal for Example 6.49.

In the obtained polynomial invariant we may substitute the concrete values for the symbolically treated initial values $d[0], r[0], rp[0], q[0]$ given at the entry of the loop. By further algebraic manipulations based on additional `Mod` and `Quotient` properties, we thus obtain

$$\{2\ d[0]\ q[0] - d[0]^2\ q[0] - 8\ r[0] + 4\ d[0]\ r[0] - 4\ d[0]\ rp[0] = -8\ n\}.$$

Hence the output of `Aligator`, by *user guidance*, could be simplified to

$$\{\ 2\ d\ q - d^2\ q - 8\ r + 4\ d\ r - 4\ d\ rp = -8\ n\ \}.$$

However, let us emphasize again, that the polynomial invariant with symbolic initial values is automatically inferred by our software package `Aligator`.

Verification in Theorema

The considered imperative loop is taken from (Knuth, 1998; Rodriguez-Carbonell and Kapur, 2007a) and implements an algorithm searching for the smallest odd factor d of a given positive

odd large integer n, using an odd parameter f such that $f < d \wedge d^2 \leq n < (d+1)^2$ with $f \geq 2\sqrt[3]{n}+1$.

In the imperative verification environment of *Theorema* (see Section 3), the specification of the algorithm is as follows.

$$\begin{aligned}&\texttt{Specification}[\text{"}\textit{Factor}\text{"},\ \textit{Factor}[\downarrow n, \downarrow f, \uparrow d]\\ &\quad \texttt{Pre} \rightarrow ((\texttt{Mod}[n,2]=1) \wedge (n>0) \wedge (\texttt{Mod}[f,2]=1) \wedge ((f-1)^3 \geq 8*n)),\\ &\quad \texttt{Post} \rightarrow ((r=0) \implies ((\texttt{Mod}[n,d]=0) \wedge (\texttt{Mod}[d,2]=1) \wedge (f<d) \wedge (d^2 \leq n < (d+1)^2)))].\end{aligned}$$

Using the (simplified) polynomial invariant generated by `Aligator` and other non-polynomial invariant properties, the annotated program code is given below.

$$\begin{aligned}&\texttt{Program}[\text{"}\textit{Factor}\text{"},\ \textit{Factor}[\downarrow n, \downarrow f, \uparrow d]\\ &\texttt{Module}[\{r,rp,q,s,t\},\\ &\ s := \texttt{Sqrt}[n];\\ &\ d := f; r := \texttt{Mod}[n,d]; rp := \texttt{Mod}[n,d-2];\\ &\ q := 4*(\texttt{Quotient}[n,d-2] - \texttt{Quotient}[n,d]);\\ &\ \texttt{WHILE}[(s \geq d) \wedge (r \neq 0),\\ &\quad \texttt{IF}[2*r-rp+q<0,\\ &\qquad t := r; r := 2*r-rp+q+d+2; rp := t; q := q+4; d := d+2,\\ &\qquad \texttt{IF}[(2*r-rp+q \geq 0) \wedge (2*r-rp+q < d+2),\\ &\qquad\quad t := r; r := 2*r-rp+q; rp := t; d := d+2,\\ &\qquad\quad \texttt{IF}[(2*r-rp+q \geq 0) \wedge (2*r-rp+q \geq d+2) \wedge (2*r-rp+q < 2*d+4),\\ &\qquad\qquad t := r; r := 2*r-rp+q-d-2; rp := t; q := q-4; d := d+2,\\ &\qquad\qquad t := r; r := 2*r-rp+q-2*d-4; rp := t; q := q-8; d := d+2]]],\\ &\quad \texttt{Invariant} \rightarrow (d^2*q-4*r*d+4*rp*d-2*d*q+8*r = 8*n) \wedge (\texttt{Mod}[d,2]=1)]]].\end{aligned}$$

Calling the `VCG` for verifying the partial correctness of the "`Factor`" program w.r.t. its given specification, namely:

$$\texttt{VCG}[\texttt{Program}[\text{"}\textit{Factor}\text{"}], \texttt{Specification}[\text{"}\textit{Factor}\text{"}]],$$

we obtain 3 universally quantified verification conditions as a *Theorema* lemma, expressed in first order predicate logic.

Lemma (Factor) :

for any : n, f, d, r, rp, q, s, t

(WHILE.Inv)

$$\begin{aligned}&(8*r+(-2)*q*d+(-4)*r*d+4*\text{rp}*d+q*d^2 = 8*n) \bigwedge (\text{Mod}[d,\ 2]=1) \bigwedge s \geq d \bigwedge r \neq 0 \Rightarrow\\ &\quad (q+2*r+(-1)*\text{rp}<0 \Rightarrow\\ &\qquad (-2*(4+q)*(2+d)+4*r*(2+d)+(4+q)*(2+d)^2+8*(2+q+2*r+(-1)*\text{rp}+d)+\\ &\qquad\quad (-4)*(2+d)*(2+q+2*r+(-1)*\text{rp}+d) = 8*n) \bigwedge (\text{Mod}[2+d,\ 2]=1)) \bigwedge (q+2*r+(-1)*\text{rp} \geq 0 \Rightarrow\end{aligned}$$

$$
\begin{array}{l}
(q+2*r+(-1)*\mathrm{rp}\geq 0 \wedge q+2*r+(-1)*\mathrm{rp}<2+\boldsymbol{d} \Rightarrow \\
\quad (8*(q+2*r+(-1)*\mathrm{rp})+(-2)*q*(2+\boldsymbol{d})+4*r*(2+\boldsymbol{d})+ \\
\qquad (-4)*(q+2*r+(-1)*\mathrm{rp})*(2+\boldsymbol{d})+q*(2+\boldsymbol{d})^2=8*\boldsymbol{n}) \bigwedge (\mathrm{Mod}[2+\boldsymbol{d},2]=1)) \bigwedge \\
(\neg\,(q+2*r+(-1)*\mathrm{rp}\geq 0 \wedge q+2*r+(-1)*\mathrm{rp}<2+\boldsymbol{d}) \Rightarrow \\
\quad (q+2*r+(-1)*\mathrm{rp}\geq 0 \wedge q+2*r+(-1)*\mathrm{rp}\geq 2+\boldsymbol{d} \wedge q+2*r+(-1)*\mathrm{rp}<4+2*\boldsymbol{d} \Rightarrow \\
\qquad (8*(-2+q+2*r+(-1)*\mathrm{rp}+(-1)*\boldsymbol{d})+(-2)*(-4+q)*(2+\boldsymbol{d})+4*r*(2+\boldsymbol{d})+ \\
\qquad\quad (-4)*(-2+q+2*r+(-1)*\mathrm{rp}+(-1)*\boldsymbol{d})*(2+\boldsymbol{d})+(-4+q)*(2+\boldsymbol{d})^2=8*\boldsymbol{n}) \bigwedge \\
\qquad (\mathrm{Mod}[2+\boldsymbol{d},2]=1)) \bigwedge \\
(\neg\,(q+2*r+(-1)*\mathrm{rp}\geq 0 \wedge q+2*r+(-1)*\mathrm{rp}\geq 2+\boldsymbol{d} \wedge q+2*r+(-1)*\mathrm{rp}<4+2*\boldsymbol{d}) \Rightarrow \\
\quad (8*(-4+q+2*r+(-1)*\mathrm{rp}+(-2)*\boldsymbol{d})+(-2)*(-8+q)*(2+\boldsymbol{d})+4*r*(2+\boldsymbol{d})+ \\
\qquad (-4)*(-4+q+2*r+(-1)*\mathrm{rp}+(-2)*\boldsymbol{d})*(2+\boldsymbol{d})+(-8+q)*(2+\boldsymbol{d})^2=8*\boldsymbol{n}) \bigwedge \\
\quad (\mathrm{Mod}[2+\boldsymbol{d},2]=1))))
\end{array}
$$

(WHILE.Final)

$$
\begin{array}{l}
(8*r+(-2)*q*\boldsymbol{d}+(-4)*r*\boldsymbol{d}+4*\mathrm{rp}*\boldsymbol{d}+q*\boldsymbol{d}^2=8*\boldsymbol{n}) \bigwedge (\mathrm{Mod}[\boldsymbol{d},2]=1) \bigwedge \neg\,(s\geq \boldsymbol{d} \wedge r\neq 0) \Rightarrow \\
\quad ((r=0) \Rightarrow (\mathrm{Mod}[\boldsymbol{n},\boldsymbol{d}]=0) \bigwedge (\mathrm{Mod}[\boldsymbol{d},2]=1) \bigwedge \boldsymbol{f}<\boldsymbol{d} \bigwedge \boldsymbol{d}^2\leq \boldsymbol{n} \bigwedge \boldsymbol{n}<(1+\boldsymbol{d})^2)
\end{array}
$$

(Init)

$$
\begin{array}{l}
(\mathrm{Mod}[\boldsymbol{n},2]=1) \bigwedge \boldsymbol{n}>0 \bigwedge (\mathrm{Mod}[\boldsymbol{f},2]=1) \bigwedge (-1+\boldsymbol{f})^3\geq 8*\boldsymbol{n} \Rightarrow \\
\quad (8*(\mathrm{Mod}[\boldsymbol{n},\boldsymbol{f}])+4*(\mathrm{Mod}[\boldsymbol{n},-2+\boldsymbol{f}])*\boldsymbol{f}+(-4)*(\mathrm{Mod}[\boldsymbol{n},\boldsymbol{f}])*\boldsymbol{f}+ \\
\qquad (-8)*(\mathrm{Quotient}[\boldsymbol{n},-2+\boldsymbol{f}]+(-1)*\mathrm{Quotient}[\boldsymbol{n},\boldsymbol{f}])*\boldsymbol{f}+ \\
\qquad 4*(\mathrm{Quotient}[\boldsymbol{n},-2+\boldsymbol{f}]+(-1)*\mathrm{Quotient}[\boldsymbol{n},\boldsymbol{f}])*\boldsymbol{f}^2=8*\boldsymbol{n}) \bigwedge (\mathrm{Mod}[\boldsymbol{f},2]=1)
\end{array}
$$

Example 6.50 Given the P-solvable imperative loop with conditional branches and assignments:

$$
\begin{array}{l}
a:=x;b:=y;p:=1;q:=0; \\
\texttt{While}[((a\neq 0)\wedge(b\neq 0)), \\
\texttt{If}[((\texttt{Mod}[a,2]=0)\wedge(\texttt{Mod}[b,2]=0)) \\
\quad \texttt{Then}\ a:=a/2;b:=b/2;p:=4*p \\
\quad \texttt{Else If}[((\texttt{Mod}[a,2]=1)\wedge(\texttt{Mod}[b,2]=0)) \\
\quad \texttt{Then}\ a:=a-1;q:=q+b*p \\
\quad \texttt{Else If}[((\texttt{Mod}[a,2]=0)\wedge(\texttt{Mod}[b,2]=1)) \\
\quad \texttt{Then}\ b:=b-1;q:=q+a*p \\
\quad \texttt{Else}\ q:=q+(a+b-1)*p;a:=a-1;b:=b-1]]]].
\end{array}
$$

By applying Algorithm 6.32 and using `Aligator`, the obtained polynomial invariant is

$$\{\ abp-a[0]\ b[0]\ p[0]+q-q[0]\ \}.$$

Moreover, by the third condition of Theorem 6.40, we conclude that the polynomial thus obtained generates the ideal of all polynomial invariants for Example 6.50.

Further, in the obtained polynomial invariant we may substitute the concrete values for the symbolically treated initial values $a[0], b[0], p[0], q[0]$ given at the entry of the loop. We thus obtain

$$\{\, abp + q - xy \,\}.$$

Verification in Theorema

The considered imperative loop is taken from (Rodriguez-Carbonell and Kapur, 2007b), and implements another version of an algorithm computing the product q of two integers $x,\ y$.

In the imperative verification environment of *Theorema* (see Section 3), the specification of the algorithm is as follows.

$$\begin{aligned}&\texttt{Specification}[\text{"}\textit{Product}\text{"},\ \textit{Product}[\downarrow x, \downarrow y, \uparrow q]\\ &\quad \texttt{Pre} \rightarrow ((x \geq 0) \wedge (y > 0)),\\ &\quad \texttt{Post} \rightarrow (q = x * y)].\end{aligned}$$

Using the polynomial invariant generated by `Aligator`, the annotated program code is given below.

$$\begin{aligned}&\texttt{Program}[\text{"}\textit{Product}\text{"},\ \textit{Product}[\downarrow x, \downarrow y, \uparrow q]\\ &\texttt{Module}[\{a, b, p\},\\ &a := x; b := y; p := 1; q := 0;\\ &\ \texttt{WHILE}[(a \neq 0) \wedge (b \neq 0),\\ &\quad \texttt{IF}[(\texttt{Mod}[a, 2] = 0) \wedge (\texttt{Mod}[b, 2] = 0),\\ &\qquad a := a/2; b := b/2; p := 4 * p,\\ &\qquad \texttt{IF}[(\texttt{Mod}[a, 2] = 1) \wedge (\texttt{Mod}[b, 2] = 0)\\ &\qquad\quad a := a - 1; q := q + b * p,\\ &\qquad\quad \texttt{IF}[(\texttt{Mod}[a, 2] = 0) \wedge (\texttt{Mod}[b, 2] = 1),\\ &\qquad\qquad b := b - 1; q := q + a * p,\\ &\qquad\qquad q := q + (a + b - 1) * p; a := a - 1; b := b - 1]]],\\ &\quad \texttt{Invariant} \rightarrow (a * b * p + q = x * y)]]].\end{aligned}$$

Calling the `VCG` for verifying the partial correctness of the "`Product`" program w.r.t. its given specification, namely:

$$\texttt{VCG}[\texttt{Program}[\text{"}\textit{Product}\text{"}], \texttt{Specification}[\text{"}\textit{Product}\text{"}]],$$

we obtain 3 universally quantified verification conditions as a *Theorema* lemma, expressed in first order predicate logic, as follows.

Lemma (Product) :
for any : x, y, q, a, b, p
(WHILE.Inv)
$(a * b * p + q = x * y) \wedge a \neq 0 \wedge b \neq 0 \Rightarrow ((\text{Mod}[a, 2] = 0) \wedge (\text{Mod}[b, 2] = 0) \Rightarrow (a * b * p + q = x * y)) \wedge$

$$(\neg\,((\mathrm{Mod}[a,\,2] = 0) \wedge (\mathrm{Mod}[b,\,2] = 0)) \Rightarrow$$
$$((\mathrm{Mod}[a,\,2] = 1) \wedge (\mathrm{Mod}[b,\,2] = 0) \Rightarrow (b * p + (-1 + a) * b * p + \boldsymbol{q} = \boldsymbol{x} * \boldsymbol{y})) \wedge$$
$$(\neg\,((\mathrm{Mod}[a,\,2] = 1) \wedge (\mathrm{Mod}[b,\,2] = 0)) \Rightarrow$$
$$((\mathrm{Mod}[a,\,2] = 0) \wedge (\mathrm{Mod}[b,\,2] = 1) \Rightarrow (a * p + a * (-1 + b) * p + \boldsymbol{q} = \boldsymbol{x} * \boldsymbol{y})) \wedge$$
$$(\neg\,((\mathrm{Mod}[a,\,2] = 0) \wedge (\mathrm{Mod}[b,\,2] = 1)) \Rightarrow ((-1 + a) * (-1 + b) * p + (-1 + a + b) * p + \boldsymbol{q} = \boldsymbol{x} * \boldsymbol{y}))))$$

(WHILE.Final)

$$(a * b * p + \boldsymbol{q} = \boldsymbol{x} * \boldsymbol{y}) \wedge \neg\,(a \neq 0 \wedge b \neq 0) \Rightarrow (\boldsymbol{q} = \boldsymbol{x} * \boldsymbol{y})$$

(Init)

$$\boldsymbol{x} \geq 0 \wedge \boldsymbol{y} > 0 \Rightarrow (\boldsymbol{x} * \boldsymbol{y} = \boldsymbol{x} * \boldsymbol{y})$$

7 Software: The `Aligator` Package

As mentioned and illustrated already on numerous examples, our approach for automated invariant generation by algebraic techniques is implemented as a *Mathematica* package. The package is called `Aligator`, and it was implemented in *Mathematica* 5.2.

`Aligator` offers software support for experimenting with the algorithms presented in this thesis for **a**utomated **l**oop **i**nvariant **g**eneration by **a**lgebraic **t**echniques **o**ver the **r**ationals. `Aligator` has been successfully tried on many programs implementing interesting algorithms working on numbers.

In this section we give a brief presentation of the current version of our software package. The interested reader may contact the author for further information. The syntax of the commands listed in this section might change in future versions of `Aligator`, due to possible improvements and speed-ups on the current implementation.

7.1 Usage of `Aligator`

In this subsection, we present the usage of our package on concrete examples, stating this way also the syntax and semantics of the top-level `Aligator` command.

Command 7.1: `Aligator[PLoopExpression, IniVal→ list of assignments]`

Input: `PLoopExpression`, that is a P-solvable while loop with only conditional branches and assignments

Output: A set of polynomial invariants $\{p_1(X), \ldots, p_r(X)\}$ among the loop variables X with initial values X_0

Optional: `IniVal→list of assignments`, specifying the initial values X_0 of loop variables

The considered loops have similar syntax and semantics as in *Mathematica*, namely:

`WHILE[test_cond, loop_body],`

where

- `test_cond` can be ... or a boolean expression. In each case, as mentioned already, test-conditions are ignored in the invariant generation process;
- `loop_body` is a sequence of assignments and conditional branches;

- assignments are usual *Mathematica* assignments of the form `x:=expression`, where `x` is a loop variable;
- conditional statements are similar to *Mathematica*'s conditionals commands, namely:

  `IF[test_cond, then_branch, else_branch]`.

  Again, `test_cond` can be ... or a boolean expression, whereas the conditional branches are sequences of assignments.

Observe, that we use the symbols `WHILE` and `IF` for denoting loop and conditional statements, in order to avoid any conflict with *Mathematica*'s `While` and `If` commands.

Example 7.2 The appropriate call for Example 5.14 is

$$\texttt{Aligator[WHILE[}y>0,\ x:=2*x; y:=\frac{1}{2}*y-1\texttt{]},\ \texttt{IniVal} \rightarrow \{x:=10; y:=10\}\texttt{]},$$

yielding

```
{-120 + 2 x + x y}
```

In case of not specifying the initial values, we write

$$\texttt{Aligator[WHILE[}y>0,\ x:=2*x; y:=\frac{1}{2}*y-1\texttt{]]},$$

and we obtain

```
{2 x + x y - 2 x[0] - x[0] y[0]}
```

In the current version of `Aligator`, its output is a set of polynomial invariants $\{p_1(X), \ldots, p_r(X)\}$ among the loop variables X with initial values X_0. From this set, the polynomial invariant equations used further in the verification process as loop invariants are directly constructed as $p_i(X) = 0$.

Example 7.3 The appropriate call for Example 5.27 is

$$\texttt{Aligator[WHILE[}\ldots,\ a:=a+1; b:=b+c; c:=c+2; s:=s+2*a+1\texttt{]},$$
$$\texttt{IniVal} \rightarrow \{a:=0; b:=0; c:=1; s:=0\}\texttt{]},$$

yielding

```
{-3 + 2 c + c^2 - 4 s, -1 + b + c - s, 1 + 2 a - c}
```

In case of not specifying the initial values, we write

$$\texttt{Aligator[WHILE[}\ldots,\ a:=a+1; b:=b+c; c:=c+2; s:=s+2*a+1\texttt{]]},$$

and we obtain

```
{-4 c - c^2 + 4 s - 4 c a[0] + 4 c[0] + 2 c c[0] + 4 a[0] c[0] - c[0]^2 - 4 s[0],
 2 b + 3 c - 2 s + 2 c a[0] - 2 b[0] - 3 c[0] - c c[0] - 2 a[0] c[0] + c[0]^2 + 2 s[0],
 2 a - c - 2 a[0] + c[0]}
```

Example 7.4 The appropriate call for Example 6.42 is

$$\begin{aligned}&\texttt{Aligator}[\\ &\qquad \texttt{WHILE}[(2*p*r \geq err),\\ &\qquad\qquad \texttt{IF}[2*r-2*q*p \geq 0,\ r := 2*r-2*q-p;\ q := q+p;\ p := p/2,\\ &\qquad\qquad\qquad\qquad r := 2*r;\ p := p/2]],\\ &\qquad \texttt{IniVal} \rightarrow \{r := a-1; q := 1, p := 1/2\}],\end{aligned}$$

yielding

```
{-a + q^2 + 2 p r}
```

In case of not specifying the initial values, we write

$$\begin{aligned}&\texttt{Aligator}[\\ &\qquad \texttt{WHILE}[(2*p*r \geq err),\\ &\qquad\qquad \texttt{IF}[2*r-2*q*p \geq 0,\ r := 2*r-2*q-p;\ q := q+p;\ p := p/2,\\ &\qquad\qquad\qquad\qquad r := 2*r;\ p := p/2]]],\end{aligned}$$

and we obtain

```
{q^2 - q[0]^2 + 2 p r - 2 p[0] r[0]}
```

The outputs of `Aligator` on Examples 7.2 and 7.3 were produced by applying Algorithm 5.11, whereas for Example 7.4 the `Aligator` result was obtained by an implementation of Algorithm 6.32.

As proved in Section 5 for P-solvable loops with assignments only, Algorithm 5.11 is complete. Thus, for P-solvable loops with assignments only, the output of `Aligator` is a basis for the polynomial invariant ideal of the loop.

Further, as discussed in Section 6, under additional assumptions presented in Subsection 6.5, Algorithm 6.32 is also complete. Namely, under the assumptions from Subsection 6.5, `Aligator` returns a basis of the polynomial invariant ideal also for some special cases of P-solvable loops with conditional branches and assignments.

For all examples treated in the thesis, `Aligator` returns a complete set of polynomial invariants from which any other polynomial invariant can be derived.

In what follows, we briefly discuss the main commands integrated in `Aligator`, implementing the steps of the above mentioned two algorithms.

7.2 Loop Transformations

Command 7.5: `CheckIfSeq[PLoopExpression]`

Input: `PLoopExpression`, that is a while loop with only conditional branches and assignments

Output: `True`, if the loop body contains conditional branches and `False`, otherwise

In the `PLoopExpression`, as well as in the conditional statements, test-conditions can be ... or boolean expressions.

Example 7.6 The appropriate call

- for Example 6.42 is

  $$\texttt{CheckIfSeq}[\texttt{WHILE}[(2*p*r \geq err),\ \texttt{IF}[2*r-2*q*p \geq 0,\ r := 2*r-2*q-p;\ q := q+p;\ p := p/2,\ r := 2*r;\ p := p/2]]],$$

  yielding `True` as output;

- for Example 5.14 is:

  $$\texttt{CheckIfSeq}[\texttt{WHILE}[y > 0,\ x := 2*x; y := \frac{1}{2}*y-1]],$$

  yielding `False` as output.

Command 7.7: `IfWhileTransform[PLoopIfExpression]`

Input: `PLoopIfExpression`, that is a while loop with only $k \geq 1$ conditional branches and assignments

Output: A list $\{S_1,\ldots,S_k\}$ of assignment sequences, where S_i is the sequence of assignments from the ith conditional branch

Command `IfWhileTransform` is implemented based on the transformation rules given by Theorems 6.1 and Theorem 6.6. Thus, its output is the list of inner loop bodies S_i, $i = 1,\ldots,k$, after applying Theorem 6.6.

Example 7.8 The appropriate call

- for Example 6.42 is

  $$\texttt{IfWhileTransform}[\texttt{WHILE}[(2*p*r \geq err),\ \texttt{IF}[2*r-2*q*p \geq 0,\ r := 2*r-2*q-p;\ q := q+p;\ p := p/2,\ r := 2*r;\ p := p/2]]],$$

yielding

```
{{r := 2 * r - 2 * q - p; q := q + p; p := p / 2}, {r := 2 * r; p := p / 2}}
```

- for Example 6.44 is

$$\begin{aligned}&\texttt{IfWhileTransform}[\\&\quad\texttt{WHILE}[(y \neq 0),\\&\qquad\texttt{IF}[(\text{Mod}[y,2] = 1),\ z := z + x; y := y - 1];\\&\qquad x := 2 * x; y := y/2]],\end{aligned}$$

yielding

```
{{z := z + x; y := y - 1; x := 2 * x; y := y / 2}, {x := 2 * x; y := y / 2}}
```

7.3 Extracting System of Recurrences

Command 7.9: `RecSystem[{assignments_sequence}]`

Input: `assignments_sequence`, that is a "flattened" loop body, namely, a sequence of assignment statements of the form $x_1 := expression_1; \ldots; x_m := expression_m$

Output: {{`RecVar`}, `VarList`, `recSystem`}, where

- `RecVar` is a fresh new variable representing the recurrence index (i.e. loop counter);
- `VarList` is a list of 2-elements lists of the form {`x[RecVar],x`} representing the correspondence between the loop variable x and its value at the `RecVar` iteration of the loop;
- `recSystem` is the system of recurrence equations of the loop body.

Example 7.10 For Example 7.2 (or equivalently for Example 5.14) the system of recurrence equations of the loop is obtained by the command call below.

$$\texttt{RecSystem}[\{x := 2 * x; y := \frac{1}{2} * y - 1\}],$$

yielding

```
{
 {x[n + 1] == 2 * x[n], y[n + 1] == 1 / 2 * y[n] - 1},
 {{x[n], x}, {y[n], y}},
 {n}
}
```

For simplicity of reading, we replaced the newly generated variable standing for the recurrence index by n.

Example 7.11 For Example 5.23 the system of recurrence equations of the loop is obtained by the command call below.

$$\texttt{RecSystem}[\{t := r;\ r := 2*r - 8*q;\ q := t;\ x := 8*x\}],$$

yielding

```
{
 {r[n + 2] == 2 * r[n + 1] - 8 * r[n], x[n + 1] == 8 * x[n]},
 {{r[n + 1], r}, {r[n], q}, {x[n], x}},
 {n}
}
```

For simplicity of reading, we replaced the newly generated variable standing for the recurrence index by n.

Example 7.12 For the first sequence of assignments from the `IfWhileTransform` command's output on Example 6.44, the system of recurrence equations of the loop is obtained by the command call below.

$$\texttt{RecSystem}[\{z := z + x;\ y := y - 1;\ x := 2*x;\ y := y/2\}],$$

yielding

```
{
 {z[n + 1] == z[n] + x[n], x[n + 1] == 2 * x[n], y[n + 1] == 1 / 2 * y[n] - 1 / 2},
 {{z[n], z}, {x[n], x}, {y[n], y}},
 {n}
}
```

For simplicity of reading, we replaced the newly generated variable standing for the recurrence index by n.

7.4 Recurrence Solving

Command 7.13: `RecSolve[RecSystem, VarList, {RecVar}]`

Input: `RecSystem`, that is a list of recurrence equations with recurrence index `RecVar`, whereas `VarList` is as presented in Command 7.9. The recurrence equations of `RecSystem` define the sequences x_i[`RecVar`] from `VarList`

Output: {`CFSystem`, {`RecVar`}, `ExpVars`, `FinVars`, `IniVars`, `AlgDep`}, where

- `CFSystem` is the closed form system of `RecSystem`. Each closed form expression is a polynomial in the initial values, `RecVar` and algebraically related exponential sequences in `RecVar`. Each exponential sequence in `RecVar` is renamed by a fresh new variable;

- `ExpVars` is the list of fresh variables introduced for the exponential sequences from the closed form expressions;
- `FinVars` is the list of variables whose closed forms are computed;
- `IniVars` is the list of initial values present in the closed form expressions;
- `AlgDep` is the set of generators for the ideal of algebraic dependencies of the exponential sequences from `ExpVars`.

Example 7.14 For Example 7.2 (or equivalently for Example 7.10) the system of its closed form solutions is obtained by the command call

$$\texttt{RecSolve}[\{x[n+1] == 2*x[n], y[n+1] == 1/2*y[n]-1\},\ \{\{x[n],x\},\{y[n],y\}\},\ \{n\}],$$

yielding

```
{
 {x == a * x[0], y == b * (y[0] - 2) + 2},
 {n}, {a, b}, {x, y}, {x[0], y[0]},
 {a * b - 1 == 0}
}
```

`RecSolve` is an implementation of Algorithms 5.12 and 4.45 , and thus it needs to incorporate special commands for solving different types of recurrences, based on the syntactic structure of the recurrence relations. This is achieved by using several *Mathematica* packages written by the RISC Combinatorics group, as follows.

- For solving Gosper-summable recurrences, more precisely to compute the sum of a hypergeometric term based on Algorithm 4.20, we rely on the *Mathematica* package called `zb`, implemented by the RISC Combinatorics group (Paule and Schorn, 1995). This is presented in Command 7.15.
- For solving C-finite recurrences based on Algorithm 4.30, we rely on the *Mathematica* package called `SumCracker`, implemented by the RISC Combinatorics group (Kauers, 2006). For solving such recurrences see Command 7.17.
- For detecting the algebraic dependencies among the algebraic exponential sequences corresponding to the variables `ExpVars` based on Algorithm 4.45, we rely on the *Mathematica* package called `Dependencies`, implemented by the RISC Combinatorics group (Kauers and Zimmermann, 2006). For details see Command 7.19.

Command 7.15: `Gosper[summand,{k,lowerBound,upperBound}]`

Input: `summand`, that is a hypergeometric term `f(k)`

Output: Closed form solution of $\sum_{k=lowerBound}^{upperBound} f(k)$ or $\{\}$ if a closed form does not exists

Example 7.16 The closed forms of the following hypergeometric sums are obtained as follows.

- $\sum_{k=0}^{n-1} k^2 * 3^k$:

$$\texttt{Gosper}[k^2 * 3^k, \{k, 0, n-1\}]$$

$$\left\{ \sum_{k=0}^{-1+n} 3^k\, k^2 == -\frac{3}{2} + \frac{1}{2}\, 3^n\, (3 - 3\,n + n^2) \right\}$$

- $\sum_{k=0}^{n-1} 2^{-k} * p[0]$, where $p[0]$ is a constant value (i.e. the sum corresponds to the hypergeometric sum from the recurrence of q from Example 6.42):

$$\texttt{Gosper}[2^(-k) * p[0], \{k, 0, n-1\}]$$

$$\left\{ \sum_{k=0}^{-1+n} 2^{-k}\, \texttt{p[0]} == 2\, \texttt{p[0]} - 2^{1-n}\, \texttt{p[0]} \right\}$$

Command 7.17: `Crack[sequence,Where`$\rightarrow${`RecEq`}`,Into`$\rightarrow${`Vars`}`]`

Input: `sequence`, that is a (C-finite) sequence $x[n]$ defined by its recurrence equation `RecEq`

Output: Closed form solution of the `sequence` in terms of the variables `Vars`

Example 7.18 For the following C-finite sequences, we obtain their closed forms as follows.

- $x[n+2] == 2x[n+1] - x[n] - 4n$:

$$\texttt{Crack}[x[n], \texttt{Where} \rightarrow \{x[n+2] == 2x[n+1] - x[n] - 4n\}, \texttt{Into} \rightarrow \{n, 2^n\}, \texttt{Degree} \rightarrow \texttt{Infinity}]$$

$$\frac{1}{3}\, (-4\,n + 6\,n^2 - 2\,n^3 + 3\,\texttt{x[0]} - 3\,n\,\texttt{x[0]} + 3\,n\,\texttt{x[1]})$$

- $p[n+1] == \frac{1}{2} * p[n]$ (i.e. the recurrence of p from Example 6.42):

$$\texttt{Crack}[p[n], \texttt{Where} \rightarrow \{p[n+1] == 1/2 * p[n]\}, \texttt{Into} \rightarrow \{n, 2^{-n}\}, \texttt{Degree} \rightarrow \texttt{Infinity}]$$

$$2^{-n}\, \texttt{p[0]}$$

- $r[n+1] == 2 * r[n] - 2 * (q[0] + 2 * p[0] - 2^{1-n} * p[0]) - 2^{-n} * p[0]$, where $p[0]$, $q[0]$ are constant values (i.e. the recurrence of r from Example 6.42):

$$\texttt{Crack}[r[n], \texttt{Where} \rightarrow \{r[n+1] == 2 * r[n] - 2 * (q[0] + 2 * p[0] - 2^{1-n} * p[0]) - 2^{-n} * p[0]\}, \texttt{Into} \rightarrow \{n, 2^n\}, \texttt{Degree} \rightarrow \texttt{Infinity}]$$

```
-2^(-1-n) (4 p[0] + 3 2^(2 n) p[0] - 2^(3+n) p[0] - 2^(2+n) q[0] + 2^(1+2 n) q[0] - 2^(2 n) r[1])
```

where

```
r[1] == 2 * r[0] - 2 * q[0] - p[0]
```

Command 7.19: `Dependencies[ExpSeq, identifier, Variable→{RecVar}]`

Input: `ExpSeq`, that is a list of algebraic exponential sequences in the recurrence index `RecVar`. `identifier` is a new symbol for representing the exponential sequences from `ExpSeq`. If `identifier` is `x`, then `x[i]` refers to the i^{th} sequence from `ExpSeq`

Output: Generators of the ideal of algebraic dependencies among `ExpSeq`

Example 7.20 The algebraic dependencies among the exponential sequences $\left(\frac{1+\sqrt{5}}{2}\right)^n$ and $\left(\frac{1-\sqrt{5}}{2}\right)^n$ are computed as follows.

$$\texttt{Dependencies}[\{(\frac{1+\sqrt{5}}{2})^n, (\frac{1-\sqrt{5}}{2})^n\}, x, \texttt{Variables} \rightarrow \{n\}]$$

```
{-1 + x[1]^2 x[2]^2}
```

Example 7.21 The algebraic dependencies among the exponential sequences 2^n and 4^n are computed as follows.

$$\texttt{Dependencies}[\{2^n, 4^n\}, x, \texttt{Variables} \rightarrow \{n\}]$$

```
{x[1]^2 - x[2]}
```

7.5 Merging Closed Form Systems

Command 7.22: `CFMerge[list_of_expressions ]`

Input: `list_of_expressions`=$\{$`expression`$_1$`, ..., expression`$_k\}$, with $k \geq 2$. Each *expression*$_i$ is a list $\{$`CFSystem`$_i, \{$`RecVar`$_i\}$, `ExpVars`$_i$, `FinVars`$_i$, `IniVars`$_i$, `AlgDep`$_i\}$ with corresponding semantics as described at Command 7.13

Output: {`MergedSystem, RecVars, ExpVars, FinVars, IniVars, AlgDep`}, where

- `MergedSystem` is the list of merged closed forms of loop variables;
- `RecVars` is the list of recurrence indexes, i.e. $\{$`RecVar`$_1, \ldots,$ `RecVar`$_k\}$;
- `ExpVars` is the list of newly introduced variables standing for the exponential sequences from the closed form;

- `FinVars` is the list of variables whose merged closed forms are computed;
- `IniVars` is the list of initial values present in the closed form expressions;
- `AlgDep` is the set of generators for the ideal of algebraic dependencies among the exponential sequences from the merged closed form system.

Command 7.22 is an implementation of the merging $\asymp$ operator defined by Definitions 6.15 and 6.16.

Example 7.23 For Example 6.42, we denote

$\texttt{CF}_1 = \{\texttt{CFSystem}_1, \{\texttt{RecVar}_1\}, \texttt{ExpVars}_1, \texttt{FinVars}_1, \texttt{IniVars}_1, \texttt{AlgDep}_1\}$ for the inner loop S_1, where

$$\begin{aligned}
&\texttt{CFSystem}_1 = \{p == x*p[0],\ q == q[0]+2*p[0]-2*y*p[0],\\
&\qquad\qquad r == z*\big(r[0]-2*q[0]-2*p[0]\big)-2*t*p[0]+2*q[0]+4*p[0]\}\\
&\{\texttt{RecVar}_1\} = \{j1\},\\
&\texttt{ExpVars}_1 = \{x,y,z,t\},\ \texttt{FinVars}_1 = \{p,q,r\}, \texttt{IniVars}_1 = \{p[0],q[0],r[0]\},\\
&\texttt{AlgDep}_1 = \{x-t == 0, y-t == 0, z*t-1 == 0\}
\end{aligned}$$

and

$\texttt{CF}_2 = \{\texttt{CFSystem}_2, \{\texttt{RecVar}_2\}, \texttt{ExpVars}_2, \texttt{FinVars}_2, \texttt{IniVars}_2, \texttt{AlgDep}_2\}$ for the inner loop S_2, where

$$\begin{aligned}
&\texttt{CFSystem}_2 = \{p == v*p[0],\ q == q[0],\ r == u*r[0]\}\\
&\{\texttt{RecVar}_2\} = \{j2\},\\
&\texttt{ExpVars}_2 = \{u,v\},\ \texttt{FinVars}_2 = \{p,q,r\}, \texttt{IniVars}_2 = \{p[0],q[0],r[0]\},\\
&\texttt{AlgDep}_2 = \{u*v-1 == 0\}
\end{aligned}$$

Thus, the merged closed form system for $S_1^{j1}; S_2^{j2}$ is obtained by the command

$$\texttt{CFMerge}[\{\texttt{CF}_1, \texttt{CF}_2\}],$$

yielding

```
{{p == v x p[0], q == 2 p[0] - 2 y p[0] + q[0],
  r == u (4 p[0] - 2 t p[0] + 2 q[0] + z (-2 p[0] - 2 q[0] + r[0]))},
 {j1, j2},
 {x, y, z, t, u, v}, {p, q, r}, {p[0], q[0], r[0]},
 {-t + x == 0, -t + y == 0, -1 + t z == 0, -1 + u v == 0}}
```

7.6 Variable Elimination on Closed Form Systems

Elimination of loop counters and auxiliary variables introduced for the algebraic exponential sequences in the loop counters from the closed form system of a P-solvable loop is performed by Gröbner basis computation w.r.t. an elimination order such that the loop counters and auxiliary variables are larger then the loop variables.

For Gröbner basis computation we use the `GroebnerBasis` command of *Mathematica*. We do not present this command in the thesis, the interested reader might consult (Wolfram, 2003) for further details.

7.7 Filtering

Command 7.24: `InvCheck[set_of_polynomials, list_of_traces ]`

Input: `set_of_polynomials`, that is a list of polynomials $\{p_1,\ldots,p_r\}$. `list_of_traces`, that is a list $\{S_1,\ldots,S_k\}$ where each S_i is a sequence of assignments

Output: Set of those polynomials from `set_of_polynomials` whose conjunction is preserved (in the sense of wp) by each S_i

This command is an implementation of step 3 of Algorithm 6.32.

Example 7.25 Following notations from Example 7.8 regarding Example 6.42, for the set

$$PI_1 = \{q^2 - q[0]^2 + 2*p*r - 2*p[0]*r[0]\}$$

the polynomial invariants are obtained as follows.

$$\texttt{InvCheck}[PI_1, \{\{r := 2*r - 2*q - p;\ q := q + p;\ p := p/2\}, \{r := 2*r;\ p := p/2\}\}]$$

```
{q^2 + 2 p r - q[0]^2 - 2 p[0] r[0]}
```

Example 7.26 A more illustrative example of this command is given below for Example 6.43.

Considering the set of polynomials J_1 and J_2 as introduced in Example 6.43, a computed basis of their intersection ideal is as follows. W.l.o.g., we denote the basis by PI_1, and it contains 18 polynomials.

```
PI1 = {
   r s^2 p[0] q[0] - q r s q[0] r[0] - s^2 p[0] q[0] r[0] + r s q[0]^2 r[0] + q s q[0] r[0]^2 -
   s q[0]^2 r[0]^2 - q r s p[0] s[0] + s^2 p[0]^2 s[0] - r s p[0] q[0] s[0] + q^2 r r[0] s[0] -
   q s p[0] r[0] s[0] - q r q[0] r[0] s[0] + 3 s p[0] q[0] r[0] s[0] - q q[0] r[0]^2 s[0] +
   q[0]^2 r[0]^2 s[0] + q r p[0] s[0]^2 - 2 s p[0]^2 s[0]^2 + q p[0] r[0] s[0]^2 -
   2 p[0] q[0] r[0] s[0]^2 + p[0]^2 s[0]^3,
  -q r + p s + q[0] r[0] - p[0] s[0],
```

$$r\,s\,p[0]\,q[0] - q\,r\,q[0]\,r[0] - s\,p[0]\,q[0]\,r[0] + r\,q[0]^2\,r[0] + q\,q[0]\,r[0]^2 - q[0]^2\,r[0]^2 - q\,r\,p[0]\,s[0] + s\,p[0]^2\,s[0] - r\,p[0]\,q[0]\,s[0] + p\,q\,r[0]\,s[0] - q\,p[0]\,r[0]\,s[0] - p\,q[0]\,r[0]\,s[0] + 3\,p[0]\,q[0]\,r[0]\,s[0] + p\,p[0]\,s[0]^2 - 2\,p[0]^2\,s[0]^2,$$

$$-s\,b[0]\,p[0] + s\,a[0]\,q[0] + q\,b[0]\,r[0] - b\,q[0]\,r[0] - q\,a[0]\,s[0] + b\,p[0]\,s[0],$$

$$r\,s^2\,a[0]\,q[0] - s^2\,b[0]\,p[0]\,r[0] - b\,r\,s\,q[0]\,r[0] + r\,s\,b[0]\,q[0]\,r[0] + q\,s\,b[0]\,r[0]^2 - s\,b[0]\,q[0]\,r[0]^2 - q\,r\,s\,a[0]\,s[0] + s^2\,a[0]\,p[0]\,s[0] - r\,s\,a[0]\,q[0]\,s[0] + b\,q\,r\,r[0]\,s[0] - q\,s\,a[0]\,r[0]\,s[0] - q\,r\,b[0]\,r[0]\,s[0] + 2\,s\,b[0]\,p[0]\,r[0]\,s[0] + s\,a[0]\,q[0]\,r[0]\,s[0] - q\,b[0]\,r[0]^2\,s[0] + b[0]\,q[0]\,r[0]^2\,s[0] + q\,r\,a[0]\,s[0]^2 - 2\,s\,a[0]\,p[0]\,s[0]^2 + q\,a[0]\,r[0]\,s[0]^2 - b[0]\,p[0]\,r[0]\,s[0]^2 - a[0]\,q[0]\,r[0]\,s[0]^2 + a[0]\,p[0]\,s[0]^3,$$

$$q\,r\,s\,b[0]\,p[0] - s^2\,b[0]\,p[0]^2 - b\,r\,s\,p[0]\,q[0] + s^2\,a[0]\,p[0]\,q[0] + r\,s\,b[0]\,p[0]\,q[0] - r\,s\,a[0]\,q[0]^2 - q^2\,r\,b[0]\,r[0] + q\,s\,b[0]\,p[0]\,r[0] + b\,q\,r\,q[0]\,r[0] - q\,s\,a[0]\,q[0]\,r[0] - s\,b[0]\,p[0]\,q[0]\,r[0] + s\,a[0]\,q[0]^2\,r[0] - q\,r\,b[0]\,p[0]\,s[0] + 2\,s\,b[0]\,p[0]^2\,s[0] + q\,r\,a[0]\,q[0]\,s[0] - 2\,s\,a[0]\,p[0]\,q[0]\,s[0] - q\,b[0]\,p[0]\,r[0]\,s[0] + q\,a[0]\,q[0]\,r[0]\,s[0] + b[0]\,p[0]\,q[0]\,r[0]\,s[0] - a[0]\,q[0]^2\,r[0]\,s[0] - b[0]\,p[0]^2\,s[0]^2 + a[0]\,p[0]\,q[0]\,s[0]^2,$$

$$r\,s\,a[0]\,q[0] - s\,b[0]\,p[0]\,r[0] - b\,r\,q[0]\,r[0] + r\,b[0]\,q[0]\,r[0] + q\,b[0]\,r[0]^2 - b[0]\,q[0]\,r[0]^2 - q\,r\,a[0]\,s[0] + s\,a[0]\,p[0]\,s[0] - r\,a[0]\,q[0]\,s[0] + b\,p\,r[0]\,s[0] - q\,a[0]\,r[0]\,s[0] - p\,b[0]\,r[0]\,s[0] + b[0]\,p[0]\,r[0]\,s[0] + 2\,a[0]\,q[0]\,r[0]\,s[0] + p\,a[0]\,s[0]^2 - 2\,a[0]\,p[0]\,s[0]^2,$$

$$q\,r\,b[0]\,p[0] - s\,b[0]\,p[0]^2 - b\,r\,p[0]\,q[0] + s\,a[0]\,p[0]\,q[0] + r\,b[0]\,p[0]\,q[0] - r\,a[0]\,q[0]^2 - p\,q\,b[0]\,r[0] + q\,b[0]\,p[0]\,r[0] + b\,p\,q[0]\,r[0] - q\,a[0]\,q[0]\,r[0] - 2\,b[0]\,p[0]\,q[0]\,r[0] + 2\,a[0]\,q[0]^2\,r[0] - p\,b[0]\,p[0]\,s[0] + 2\,b[0]\,p[0]^2\,s[0] + p\,a[0]\,q[0]\,s[0] - 2\,a[0]\,p[0]\,q[0]\,s[0],$$

$$r\,s^2\,a[0]\,b[0] - b\,r\,s\,b[0]\,r[0] - s^2\,a[0]\,b[0]\,r[0] + r\,s\,b[0]^2\,r[0] + b\,s\,b[0]\,r[0]^2 - s\,b[0]^2\,r[0]^2 - b\,r\,s\,a[0]\,s[0] + s^2\,a[0]^2\,s[0] - r\,s\,a[0]\,b[0]\,s[0] + b^2\,r\,r[0]\,s[0] - b\,s\,a[0]\,r[0]\,s[0] - b\,r\,b[0]\,r[0]\,s[0] + 3\,s\,a[0]\,b[0]\,r[0]\,s[0] - b\,b[0]\,r[0]^2\,s[0] + b[0]^2\,r[0]^2\,s[0] + b\,r\,a[0]\,s[0]^2 - 2\,s\,a[0]^2\,s[0]^2 + b\,a[0]\,r[0]\,s[0]^2 - 2\,a[0]\,b[0]\,r[0]\,s[0]^2 + a[0]^2\,s[0]^3,$$

$$q\,r\,s\,a[0]\,b[0] - s^2\,a[0]\,b[0]\,p[0] - b\,r\,s\,a[0]\,q[0] + s^2\,a[0]^2\,q[0] - b\,q\,r\,b[0]\,r[0] + q\,r\,b[0]^2\,r[0] + b\,s\,b[0]\,p[0]\,r[0] - 2\,s\,b[0]^2\,p[0]\,r[0] + b^2\,r\,q[0]\,r[0] - b\,s\,a[0]\,q[0]\,r[0] - b\,r\,b[0]\,q[0]\,r[0] + 2\,s\,a[0]\,b[0]\,q[0]\,r[0] + q\,b[0]^2\,r[0]^2 - b\,b[0]\,q[0]\,r[0]^2 - q\,r\,a[0]\,b[0]\,s[0] + 2\,s\,a[0]\,b[0]\,p[0]\,s[0] + b\,r\,a[0]\,q[0]\,s[0] - 2\,s\,a[0]^2\,q[0]\,s[0] - q\,a[0]\,b[0]\,r[0]\,s[0] + b[0]^2\,p[0]\,r[0]\,s[0] + b\,a[0]\,q[0]\,r[0]\,s[0] - a[0]\,b[0]\,q[0]\,r[0]\,s[0] - a[0]\,b[0]\,p[0]\,s[0]^2 + a[0]^2\,q[0]\,s[0]^2,$$

```
  q^2 r a[0] b[0] - b q r b[0] p[0] - q s a[0] b[0] p[0] + q r b[0]^2 p[0] + b s b[0] p[0]^2 -
  2 s b[0]^2 p[0]^2 - b q r a[0] q[0] + q s a[0]^2 q[0] - q r a[0] b[0] q[0] + b^2 r p[0] q[0] -
  b s a[0] p[0] q[0] - b r b[0] p[0] q[0] + 4 s a[0] b[0] p[0] q[0] + b r a[0] q[0]^2 -
  2 s a[0]^2 q[0]^2 + q b[0]^2 p[0] r[0] - q a[0] b[0] q[0] r[0] - b b[0] p[0] q[0] r[0] +
  b a[0] q[0]^2 r[0] + b[0]^2 p[0]^2 s[0] - 2 a[0] b[0] p[0] q[0] s[0] + a[0]^2 q[0]^2 s[0],

  p q^2 a[0] b[0] - b p q b[0] p[0] - q^2 a[0] b[0] p[0] + p q b[0]^2 p[0] + b q b[0] p[0]^2 -
  q b[0]^2 p[0]^2 - b p q a[0] q[0] + q^2 a[0]^2 q[0] - p q a[0] b[0] q[0] + b^2 p p[0] q[0] -
  b q a[0] p[0] q[0] - b p b[0] p[0] q[0] + 3 q a[0] b[0] p[0] q[0] - b b[0] p[0]^2 q[0] +
  b[0]^2 p[0]^2 q[0] + b p a[0] q[0]^2 - 2 q a[0]^2 q[0]^2 + b a[0] p[0] q[0]^2 -
  2 a[0] b[0] p[0] q[0]^2 + a[0]^2 q[0]^3,

  -r b[0] p[0] + r a[0] q[0] + p b[0] r[0] - a q[0] r[0] - p a[0] s[0] + a p[0] s[0],

  -b r + a s + b[0] r[0] - a[0] s[0],

  -b p + a q + b[0] p[0] - a[0] q[0],

  r s a[0] b[0] - b r b[0] r[0] - s a[0] b[0] r[0] + r b[0]^2 r[0] + b b[0] r[0]^2 -
  b[0]^2 r[0]^2 - b r a[0] s[0] + s a[0]^2 s[0] - r a[0] b[0] s[0] + a b r[0] s[0] -
  b a[0] r[0] s[0] - a b[0] r[0] s[0] + 3 a[0] b[0] r[0] s[0] + a a[0] s[0]^2 -
  2 a[0]^2 s[0]^2,

  q r a[0] b[0] - s a[0] b[0] p[0] - b r a[0] q[0] + s a[0]^2 q[0] - b p b[0] r[0] +
  p b[0]^2 r[0] + b b[0] p[0] r[0] - b[0]^2 p[0] r[0] + a b q[0] r[0] - b a[0] q[0] r[0] -
  a b[0] q[0] r[0] + a[0] b[0] q[0] r[0] - p a[0] b[0] s[0] + 2 a[0] b[0] p[0] s[0] +
  a a[0] q[0] s[0] - 2 a[0]^2 q[0] s[0],

  p q a[0] b[0] - b p b[0] p[0] - q a[0] b[0] p[0] + p b[0]^2 p[0] + b b[0] p[0]^2 -
  b[0]^2 p[0]^2 - b p a[0] q[0] + q a[0]^2 q[0] - p a[0] b[0] q[0] + a b p[0] q[0] -
  b a[0] p[0] q[0] - a b[0] p[0] q[0] + 3 a[0] b[0] p[0] q[0] + a a[0] q[0]^2 - 2 a[0]^2 q[0]^2
}
```

Further, by the command call

$$\texttt{InvCheck}[PI_1, \{\{a := a-b; p := p-q; r := r-s\}, \{b := b-a; q := q-p; s := s-r\}\}],$$

the obtained set of polynomial invariants is

```
{-b p + a q + b[0] p[0] - a[0] q[0],
 -b r + a s + b[0] r[0] - a[0] s[0],
 -q r + p s + q[0] r[0] - p[0] s[0],
 -r b[0] p[0] + r a[0] q[0] + p b[0] r[0] - a q[0] r[0] - p a[0] s[0] + a p[0] s[0],
 -s b[0] p[0] + s a[0] q[0] + q b[0] r[0] - b q[0] r[0] - q a[0] s[0] + b p[0] s[0]}
```

7.8 Experimental Results

As it was already shown in Sections 5 and 6, we have successfully tested our method on many textbook examples implementing interesting algorithms working on numbers, as they are included in the table below.

Example	Comb. Methods	Nr.Poly.	(sec)	Compl.
P-solvable loops with assignments only				
Division (Dijkstra, 1976)	Gosper	1	0.08	yes
Integer square root (Kirchner, 1999)	Gosper	2	0.09	yes
Integer square root (Knuth, 1998)	Gosper	2	0.09	yes
Integer cubic root (Knuth, 1998)	Gosper	2	0.15	yes
Fibonacci (Kovács, 2006)	Generating Functions, Alg.Dependencies	1	0.73	yes
Sum of powers n^5 (Petter, 2004)	Gosper	1	0.07	yes
P-solvable loops with conditional branches and assignments				
Wensley's Algorithm (Wegbreit, 1974)	Gosper, C-finite, Alg.Dependencies	2	0.48	yes (Thm. 6.38)
LCM-GCD computation (Dijkstra, 1976)	Gosper	1	0.33	yes (Thm. 6.37)
Extended GCD (Knuth, 1998)	Gosper	5	2.65	yes (Thm. 6.37)
Fermat's factorization (Knuth, 1998)	Gosper	1	0.32	yes (Thm. 6.38)
Square root (Zuse, 1993)	Gosper, C-finite, Alg.Dependencies	1	1.28	yes (Thm. 6.38)
Binary Product (Knuth, 1998)	Gosper, C-finite, Alg.Dependencies	1	0.47	yes (Thm. 6.38)
Binary Product (Rodriguez-Carbonell and Kapur, 2007b)	Gosper, C-finite, Alg.Dependencies	1	9.6	yes (Thm. 6.40)
Binary Division (Kaldewaij, 1990)				
1st Loop	C-finite, Alg. Dependencies	2	0.10	yes
2nd Loop	C-finite, Gosper, Alg.Dependencies	1	0.72	yes (Thm. 6.38)
Square root (Dijkstra, 1976)				
1st Loop	C-finite, Alg. Dependencies	2	0.15	yes
2nd Loop	Gosper, C-finite, Alg. Dependencies	1	8.7	yes (Thm. 6.38)
Hardware Integer Division (Manna, 1974)				
1st Loop	C-finite, Alg. Dependencies	3	0.19	yes
2nd Loop	Gosper, C-finite, Alg.Dependencies	3	0.64	yes (Thm. 6.38)
Hardware Integer Division (Sankaranaryanan *et al.*, 2004)				
1st Loop	C-finite, Alg. Dependencies	3	0.17	yes
2nd Loop	Gosper, C-finite, Alg.Dependencies	3	0.81	yes (Thm. 6.38)
Factoring Large Numbers (Knuth, 1998)	C-finite, Gosper	1	14.4	yes (Thm. 6.40)

The first column of the table contains the name of the example, the second and third columns specify the applied combinatorial methods and the number of generated polynomial invariants

for the corresponding example, whereas the fourth column shows the timing (in seconds) needed by the implementation on a Pentium 4, 1.6GHz processor with 512 Mb RAM. The fifth columns shows whether our method was complete, namely, whether the obtained polynomial invariants generate the polynomial invariant ideal of the loop. The completeness aspect of our approach illustrated in the fifth column is relevant only for loops with conditional branches: for these example we also mention which assumption from Subsection 6.5 yields the completeness of our method.

7.9 `Aligator` and *Theorema*

As mentioned already, the polynomial invariants of a given imperative loop automatically inferred using `Aligator`, together with other (user-asserted) non-polynomial equality invariants, are used as loop annotations for the verification of a given program w.r.t. its specification in the imperative verification environment of *Theorema*.

The annotation process, i.e. integration of `Aligator` in *Theorema*, is not yet computer-supported and is currently expected to be done manually by the user. However, in the future we plan to automate the annotation process in *Theorema*, using `Aligator`'s output.

In the table below we list results of proving the verification conditions obtained from *partial correctness* verification of the examples from Subsection 7.8, using automatically generated polynomial invariants by `Aligator`, as well as user-asserted, non-polynomial invariants.

The first column of the table contains the name of the example. The second column gives the number of verification conditions generated from *partial correctness* verification, whereas the third column specifies the number of non-polynomial invariants needed additionally to `Aligator`'s output. The fourth column shows the results of proving the verification conditions by the `PCS` prover of *Theorema*(`yes` in case the proof all verification conditions succeeded, and `no` when only some of the verification conditions could be proved). The fifth column specifies whether additional knowledge (e. g. properties of `GCD, LCM`) was needed to be also used in order to perform the proof.

Example	**VCs**	**Additional assertions**	**PCS**	**Additional KB**
Division (Dijkstra, 1976)	3	3	yes	no
Integer square root (Knuth, 1998)	3	2	yes	no
Integer square root (Kirchner, 1999)	3	1	yes	no
Integer cubic root (Knuth, 1998)	3	1	yes	no
Fibonacci (Kovács, 2006)	3	1	yes	no
Sum of powers n^5 (Petter, 2004)	3	3	yes	yes
Wensley's Algorithm (Wegbreit, 1974)	3	3	yes	no
LCM-GCD computation (Dijkstra, 1976)	3	3	no	yes
Extended GCD (Knuth, 1998)	3	3	no	yes
Fermat's factorization (Knuth, 1998)	3	3	no	yes
Square root (Zuse, 1993)	3	4	yes	no
Square root (Dijkstra, 1976)	5	5	yes	no
Binary Product (Knuth, 1998)	3	3	yes	no
Binary Product (Rodriguez-Carbonell and Kapur, 2007b)	3	0	yes	no
Binary Division (Kaldewaij, 1990)	5	8	no	yes
Hardware Integer Division (Manna, 1974)	5	5	yes	no
Hardware Integer Division (Sankaranaryanan *et al.*, 2004)	5	5	yes	no
Factoring Large Numbers (Knuth, 1998)	3	1	no	yes

8 Conclusions and Further Work

In this thesis we present a framework for generating loop invariants for a family of imperative programs operating on numbers. We give several methods for invariant generation and prove a number of new results showing soundness, and also sometimes completeness of these methods. These results use non-trivial mathematics based on combining combinatorics, algebraic relations and logic. Moreover, the framework is implemented as a *Mathematica* package, called `Aligator`, and used further for imperative program verification in the *Theorema* system.

In various chapters of the thesis we present a collection of examples illustrating the power of the framework. For all of these examples a basis of polynomial invariants have been generated automatically and the generated invariants can be subsequently used for verifying properties of programs.

`Aligator` demonstrates the applicability and the usefulness of several algebraic and combinatorial techniques for automatic generation of polynomial invariants, and, in more general terms, the power of combining techniques from computational logic with techniques from computer algebra in an imperative programming environment.

For P-solvable loops having only assignments and ignored test conditions the developed algorithmic methods are showed to be complete in generating a set of polynomial invariants from which any other polynomial invariant can be derived.

Moreover, under additional assumptions for P-solvable loops with conditional branches and assignments, and all test conditions being ignored, the approach is also proved to generate a complete set of polynomial invariants of the loop. However, we could not find any example for which the our method fails to be complete. We thus conjecture that our approach covers a rich class of P-solvable loops with conditional branches and assignments, and consider the study of completeness of our approach without the additional assumptions as a challenging task for further research.

So far, the focus has been on generating polynomial equations as loop invariants. We believe that in addition to polynomial equations it should be possible to identify and generate polynomial inequalities as invariants as well. A "hidden" problem in the theoretical treatment of invariants is the fact that in most practical situations it will also contain information about other parts of the program, which is not related to the respective loop, e.g. pre- and postcondition. In (Kovács and Jebelean, 2004a) we investigated the manipulation of pre- and postconditions, and other annotations of programs, if available, along with conditions in loops and conditional statements, as well as the simple fact that no loop is executed less than 0 times. It would be interesting also to use in this context quantifier elimination methods on theories, including the theory of real closed fields.

An interesting topic for further research is the study of loops with assignment statements describ-

ing other kinds of recurrence as well, besides the C-finite and Gosper-summable case or solving using generating functions. Loops with arbitrary polynomial assignments might easily lead to more general recurrences, such as linear recurrences with polynomial coefficients in the loop counter, called P-finite. Algorithmic methods are available for solving such recurrences in terms of hypergeometric sequences (Nemes and Petkovsek, 1995). A more general class of recurrences are the so-called $\Pi\Sigma$ ones (Karr, 1981; Schneider, 2001). Handling such recurrences requires sophisticated algorithms (Schneider, 2004, 2005). Closed form solutions of such recurrences might not be polynomials in loop counters and additional variables, but still could be handled in a "polynomial manner".

Regarding the structure of loops, we plan to extend our approach by handling nested loops, as well as allowing procedure calls in the loop body.

We are also interested in generalizing the framework to programs on non-numeric data structures. More precisely, based on the algebraic approach for invariant generation presented in the thesis, we are interested in developing methods for the combination of polynomial algebra and algorithmic combinatorics with other theories such as the first-order theories of uninterpreted functions and/or arrays.

References

Adams, W. W. and Loustaunau, P. (1994). *An Introduction to Gröbner Bases*. Graduate Studies in Math. 3. AMS.

Andrews, J. H. (1998). Testing Using Log File Analysis: Tools, Methods and Issues. In *Proc. of 13th Annual Int. Conference on Automated Software Engineering (ASE'98)*.

Apt, K. R. and Olderog, E. R. (1997). *Verification of Sequential and Concurrent Programs*. Springer, 2nd edition.

Barnes, J. (2003). *High Integrity Software - The Spark Approach to Safety and Security*. Addison-Wesley.

Bensalem, S., Lakhnech, Y., and Saidi, H. (1996). Powerful Techniques for the Automatic Generation of Invariants. In *Proc. of SAS 1996*, volume 1102 of *LNCS*, pages 323–335.

Bjørner, N., Browne, A., and Manna, Z. (1997). Automatic Generation of Invariants and Intermediate Assertions. *Theoretical Computer Science*, **173**(1), 49–87.

Borwein, J. M. and Lisoněk, P. (2000). Application of Integer Relation Algorithms. *Discrete Mathematics*, **23**, 65–82.

Buchberger, B. (1985). Gröbner-Bases: An Algorithmic Method in Polynomial Ideal Theory. In *Multidimensional Systems Theory - Progress, Directions and Open Problems in Multidimensional Systems*, pages 184–232.

Buchberger, B. (1996). Symbolic computation: Computer algebra and logic. In *Frontiers of Combining Systems*, volume 3 of *Applied Logic Series*, pages 193–219. Kluwer Academic Publishers.

Buchberger, B. (1998). Introduction to Gröbner Bases. In *Gröbner Bases and Applications*, number 251 in London Mathematical Society Lecture Notes Series, pages 3–31. Cambridge University Press.

Buchberger, B. (2003). Practical and Theoretical Aspects of Program Verification. In *Computer Aided Verification of Information Systems, Romanian-Austrian Workshop*, Timişoara, Romania.

Buchberger, B. (2006). An Algorithm for Finding the Basis Elements of the Residue Class Ring of a Zero Dimensional Polynomial Ideal. *Journal of Symbolic Computation*, **41**(3-4), 475–511. Phd thesis 1965, University of Innsbruck, Austria.

Buchberger, B. and Lichtenberger, F. (1981). *Mathematics for Computer Science I - The Method of Mathematics (in German)*. Springer, 2nd edition.

Buchberger, B., Craciun, A., Jebelean, T., Kovacs, L., Kutsia, T., Nakagawa, K., Piroi, F., Popov, N., Robu, J., Rosenkranz, M., and Windsteiger, W. (2006). Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, **4**(4), 470–504.

Collins, G. E. (1975). Quantifier Elimination for the Elementary Theory of RealClosed Fields by Cylindrical Algebraic Decomposition. *LNCS*, **33**, 134–183.

Colon, M. A., Sankaranarayanan, S., and Sipma, H. B. (2003). Linear Invariant Generation Using Non-Linear Constraint Solving. In *Proc. of CAV 2003*, volume 2725 of *LNCS*, pages 420–432.

Cook, B., Podelski, A., and Rybalchenko, A. (2005). Abstraction Refinement for Termination. In *Proc. of SAS'05*, pages 87–101.

Cook, B., Podelski, A., and Rybalchenko, A. (2006). Termination Proofs for Systems Code. In *Proc. of PLDI'06*, pages 415–426.

Cook, J. E. and Wolf, A. L. (1998). Discovering Models of Software Processes from Event-based Data. *ACM Transactions on Software Engineering and Methodology*, **7**(3), 215–249.

Cousot, P. and Cousot, R. (1976). Static Determination of Dynamic Properties of Programs. In *Proc. of the 2nd International Symposium on Programming*, pages 106–130.

Cousot, P. and Cousot, R. (1977a). Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252.

Cousot, P. and Cousot, R. (1977b). Automatic Synthesis of Optimal Invariant Assertions: Mathematical Foundations. In *Proc. of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 1–12.

Cousot, P. and Cousot, R. (1979). Systematic Design of Program Analysis Frameworks. In *Conference Record of the 6th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282.

Cousot, P. and Halbwachs, N. (1978). Automatic Discovery of Linear Restraints among Variables of a Program. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97.

Cox, D., Little, J., and O'Shea, D. (1998). *Ideal, Varieties, and Algorithms. An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer, second edition.

Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall.

Elspas, B., Green, M. W., Lewitt, K. N., and Waldinger, R. J. (1972). Research in Interactive Program - Proving Techniques. Technical report, Stanford Research Institute.

Enderton, H. (1992). *Mathematical Logic, an Introduction*. Academic Press.

Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D. (2001). Dynamically Discovering Likely Program Invariants to Support Program Evaluation. *IEEE*, **TSE 27**(2).

Everest, G., van der Poorten, A., Shparlinski, I., and Ward, T. (2003). *Recurrence Sequences*, volume 104 of *Mathematical Surveys and Monographs*. American Mathematical Society.

Floyd, R. W. (1967). Assigning Meanings to Programs. In *Proc. Symphosia in Applied Mathematics 19*, pages 19–37.

Futschek, G. (1989). *Programmentwicklung und Verifikation*. Springer.

Ge, G. (1993). *Algorithms Related to Multiplicative Representation of Algebraic Numbers*. Ph.D. thesis, U. C. Berkeley.

Gerhold, S. (2002). *Uncoupling Systems of Linear Ore Operator Equations*. Master's thesis, RISC-Linz.

German, S. M. and Wegbreit, B. (1975). A Synthesizer of Inductive Assertions. In *IEEE Transactions on Software Engineering*, volume 1, pages 68–75.

Gosper, R. W. (1978). Decision Procedures for Indefinite Hypergeometric Summation. *Journal of Symbolic Computation*, **75**, 40–42.

Graham, R. L., Knuth, D. E., and Patashnik, O. (1989). *Concrete Mathematics*. Addison-Wesley Publishing Company, 2nd edition.

Gries, D. (1981). *The Science of Programming*. Springer.

Gulwani, S. and Necula, G. (2003). Discovering Affine Equalities Using Random Interpretations. In *Proc. of 30th POPL*, pages 74–84.

Hoare, C. A. R. (1969). An Axiomatic Basis for Computer Programming. *Communications of ACM*, **12**.

Jebelean, T., Kovács, L., and Popov, N. (2004). Large Experimental Program Verification in the Theorema System. In *Proc. of ISOLA 2004*, pages 92–99.

Kaldewaij, A. (1990). *Programming. The Derivation of Algorithms*. Prentince-Hall.

Kapur, D. (2006). A Quantifier Elimination based Heuristic for Automatically Generating Inductive Assertions for Programs. *Journal of Systems Science and Complexity*, **19**(3), 307–330.

Karr, M. (1976). Affine Relationships Amomg Variables of Programs. *Acta Informatica*, **6**, 133–151.

Karr, M. (1981). Summation in Finite Terms. *ACM*, **28**, 305–350.

Katz, S. and Manna, Z. (1976). Logical Analysis of Programs. *Communications of the ACM*, **19**(4), 188–206.

Kauers, M. (2005). *Algorithms for Nonlinear Higher Order Difference Equations*. Ph.D. thesis, RISC-Linz, Johannes Kepler University Linz, Austria.

Kauers, M. (2006). SumCracker: A Package for Manipulating Symbolic Sums and Related Objects. *Journal of Symbolic Computation*, **41**, 1039–1057.

Kauers, M. and Zimmermann, B. (2006). Computing the Algebraic Relations of C-finite Sequences and Multisequences. Technical Report 2006-24, SFB F013.

Kirchner, M. (1999). Program Verification with the Mathematical Software System Theorema. Technical Report 99-16, RISC-Linz, Austria. Diplomaarbeit.

Knuth, D. E. (1998). *The Art of Computer Programming*, volume 2. Seminumerical Algorithms. Addison-Wesley, 3rd edition.

Kovács, L. (2004a). Loop Verification and Imperative Program Execution in Theorema. In *Proc. of CAVIS-04*.

Kovács, L. (2004b). *Verification of Imperative Programs in Theorema*. Master's thesis, West University of Timişoara, Faculty of Mathematics and Computer Science.

Kovács, L. (2006). Finding Polynomial Invariants for Imperative Loops in the Theorema System. Technical Report 06-03, RISC-Linz, Austria.

Kovács, L. and Jebelean, T. (2003a). Generation of Invariants in Theorema. In *Proceedings of the 10th International Symphosium of Mathematics and its Application*, pages 407–415.

Kovács, L. and Jebelean, T. (2003b). Practical Aspects of Imperative Program Verification in Theorema. *Analele Universitatii din Timisoara - Proc. of SYNASC'03*, **XLI**, 135–154.

Kovács, L. and Jebelean, T. (2004a). Automated Generation of Loop Invariants by Recurrence Solving in Theorema. *Analele Universitatii din Timisoara - Proc. of SYNASC'04*, **XLII**, 151–166.

Kovács, L. and Jebelean, T. (2004b). Generation of Loop Invariants in Theorema by Combinatorial and Algebraic Methods. *Bulletins for Applied and Computer Mathematics*, **CVI**(2172), 125–134.

Kovács, L. and Jebelean, T. (2005). An Algorithm for Automated Generation of Invariants for Loops with Conditionals. *IEEE*, pages 245–249. Proc. of SYNASC'05.

Kovács, L. and Jebelean, T. (2006). Finding Polynomial Invariants for Imperative Loops in the Theorema System. In *Proc. of Verify'06, FLoC'06*, pages 52–67.

Kovács, L., Jebelean, T., and Popov, N. (2003). Verification of Imperative Programs in Theorema. In D. Dranidis and K. Tigka, editors, *Proceedings of the 1st South-East European Workshop on Formal Methods (SEEFM03)*, pages 140–147, Thessaloniki, Greece.

Kovács, L., Popov, N., and Jebelean, T. (2005a). A Verification Environment for Imperative and Functional Programs in the Theorema System. *Annals of Mathematics, Computing and Teleinformatics (AMCT), TEI Larissa, Greece*, **1**(2), 27–34.

Kovács, L., Popov, N., and Jebelean, T. (2005b). A Verification Environment for Imperative and Functional Programs in the Theorema System. In *Proc. of 2nd South-East European Workshop on Formal Methods (SEEFM05)*.

Kovács, L., Jebelean, T., and Kovacs, A. (2005c). Practical Aspects of Algebraic Invariant Generation for Loops with Conditionals. *Bulletins for Applied and Computer Mathematics*, **CVIII**(2251), 116–125.

Kovács, L., Popov, N., and Jebelean, T. (2006). Combining Logic and Algebraic Techniques for Program Verification in Theorema. In *Proc. of ISOLA 2006*.

M. Müller Olm and O. Rüthing (2001). The Complexity of Constant Propagation. In *Proc. of ESOP*, volume 2028 of *LNCS*, pages 190–205.

Mallinger, C. (1996). *Algorithmic Manipulations and Transformations of Univariate Holonomic Functions and Sequences*. Master's thesis, RISC, Johannes Kepler University, Linz.

Manna, Z. (1974). *Mathematical Theory of Computation*. McGraw-Hill Inc.

Müller-Olm, M. and Seidl, H. (2002). Polynomial Constants are Decidable. In *Proc. of SAS 2002*, volume 2477 of *LNCS*. pp. 4-19.

Müller-Olm, M. and Seidl, H. (2004a). A Note on Karr's Algorithm. In *Proc. of ICALP 2004*, volume 3142 of *LNCS*, pages 1016–1027.

Müller-Olm, M. and Seidl, H. (2004b). Computing Polynomial Program Invariants. *Indormation Processing Letters*, **91**(5), 233–244.

Müller-Olm, M. and Seidl, H. (2004c). Precise Interprocedural Analysis through Linear Algebra. In *Proc. of the 31st POPL*, pages 330–341.

Müller-Olm, M., Petter, M., and Seidl, H. (2006). Interprocedurally Analyzing Polynomial Identities. In *Proc. of STACS 2006*.

Nakagawa, K. (2002). Logico-Grafic Symbols in Theorema. Technical Report 02-60, RISC-Linz. In Proc. of Logic, Mathematics, and Computer Science (LMCS), RISC.

Nemes, I. and Petkovsek, M. (1995). RComp: A Mathematica Package for Computing with Recursive Sequences. *Journal of Symbolic Computation*, **20**(5-6), 745–753.

Paule, P. and Schorn, M. (1995). A Mathematica Version of Zeilberger's Algorithm for Proving Binomial Coefficient Identities. *Journal of Symbolic Computation*, **20**(5-6), 673–698.

Petter, M. (2004). *Berechnung von polynomiellen Invarianten*. Master's thesis, Technical University Munich, Germany.

Piroi, F. (2004). *Tools for Using Automated Provers in Mathematical Theory Exploration*. Ph.D. thesis, RISC, J. Kepler University Linz.

Robinson, A. and Voronkov, A. (2001). *Handbook of Automated Reasoning*. Elsevier Science.

Rodriguez-Carbonell, E. and Kapur, D. (2004). Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations. In *Proc. of ISSAC 04*.

Rodriguez-Carbonell, E. and Kapur, D. (2007a). Automatic Generation of Polynomial Invariants of Bounded Degree using Abstract Interpretation. *Science of Computer Programming*, **64**(1).

Rodriguez-Carbonell, E. and Kapur, D. (2007b). Generating All Polynomial Invariants in Simple Loops. *J. of Symbolic Computation*, **42**(4), 443–476.

Sankaranaryanan, S., Sipma, H. B., and Manna, Z. (2004). Non-Linear Loop Invariant Generation using Gröbner Bases. In *Proc. of POPL 2004*.

Schneider, C. (2001). *Symbolic Summation in Difference Fields*. Ph.D. thesis, RISC, J. Kepler University Linz.

Schneider, C. (2004). Symbolic Summation with Single-Nested Sum Extensions. In *Proc. ISSAC'04*, pages 282–289.

Schneider, C. (2005). Finding Telescopers with Minimal Depth for Indefinite Nested Sum and Product Expressions. In *Proc. ISSAC'05*, pages 285–292.

Tiwari, A., Ruess, H., Saidi, H., and Shankar, N. (2001). A Technique for Invariant Generation. In *Proc. of TACAS*, volume 2031 of *LNCS*, pages 113–127.

Wegbreit, B. (1974). The Synthesis of Loop Predicates. *Communication of the ACM*, **2**(17), 102–112.

Winkler, F. (1996). *Polynomial Algorithms in Computer Algebra*. Springer.

Winskel, G. (1994). *The Formal Semantics of Programming Languages. An Introduction*. The MIT Pres.

Wolfram, S. (2003). *The Mathematica Book. Version 5.0*. Wolfram Media.

Zeilberger, D. (1990). A Holonomic System Approach to Special Functions. *Journal of Computational and Applied Mathematics*, **32**, 321–368.

Zürcher, B. (1994). *Rationale Normalfomen von pseudo-linearen Abbildungen*. Master's thesis, ETH Zürich.

Zuse, K. (1993). *The Computer - My Life*. Springer.

List of Symbols

Mathematical Notation

$\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}$	The set of natural, integer, rational, real numbers
$\gcd(x,y)$	The greatest common divisor of naturals numbers x and y
$n!$	The factorial of the natural number n
$n^{\underline{a}}$	The falling factorial of the natural number n: $n^{\underline{a}} = n(n-1)\cdots(n-a+1)$, where $a \in \mathbb{N}$
$[z^n]R(z)$	The coefficient of z^n in the rational function $R(z)$
$\mathbb{K}$	Field of characteristic zero
$\bar{\mathbb{K}}$	Algebraic closure of $\mathbb{K}$
$\mathbb{K}[X] = \mathbb{K}[x_1, \ldots, x_m]$	Ring of polynomials in $x_1, \ldots, x_m$ with coefficients in $\mathbb{K}$
$I \trianglelefteq \mathbb{K}[X]$	I is an ideal of the ring $\mathbb{K}[X]$
$I = \langle p_1, \ldots, p_s \rangle$	The ideal I is generated by $p_1, \ldots, p_s$
$I + J$	Sum of ideals I and J
$I \cap J$	Intersection of ideals I and J
$\mathbb{K}[X]/I$	Quotient ring of $\mathbb{K}[X]$ by I
$p + I, p \in \mathbb{K}[X]$	Coset (element of $\mathbb{K}[X]/I$)
$\succ$	Term order
$\mathrm{LC}(p)$	Leading coefficient of the polynomial p
$\mathrm{LT}(p)$	Leading term of the polynomial p
$\mathrm{LM}(p)$	Leading monomial of the polynomial p
$\mathrm{LCM}(p,q)$	Least common multiple of polynomials p and q
$\ker\phi$	Kernel of the polynomial map ϕ

Program Verification Notation

$wp(S,Q)$	The weakest precondition of a program S with a postcondition Q
$\mathtt{vc}$	Verification conditions
$X = \{x_1,\ldots,x_m\}$	Set of loop variables with values from $\mathbb{K}$
$X_0 = \{x_{01},\ldots,x_{0m}\}$	Set of initial values of the loop variables X
$S \sqsubseteq P$	S is a possible sequence of assignments performed by P, where S is a sequence of assignments and P a basic non-deterministic program
S	Loop body with only assignments
$n \in \mathbb{N}$	Loop counter
S^n	$n \in \mathbb{N}$ times repeated execution of the loop body S
S^*	Repeated execution of the loop body S arbitrary many times
$x[n]$	The value of the loop variable x at iteration n of the loop body S, i.e. after S^n
$S_1 \vert \ldots \vert S_k$	Non-deterministic choice between $S_1,\ldots,S_k$
$x[j_1,\ldots,j_k]$	The value of the loop variable x after respectively $j_1,\ldots,j_k$ iterations of the inner loop bodies $S_1,\ldots,S_k$, i.e. after $S_1^{j_1};\ldots;S_k^{j_k}$
$E_S = \{\theta_1^n,\ldots,\theta_s^n\}$	The set of algebraic exponential sequences in the loop counter n from the P-solvable loop's closed form system
$\mathrm{CF}(S^n,E_S,X_1,X_0)$	The closed form of the P-solvable loop variables corresponding to the loop body S with only assignments, having n as the loop counter. X_1 and X_0 are the final and initial values of the loop variables.
$\mathfrak{S}_k$	The set of permutation of length k over $\{1,\ldots,k\}$
$\asymp$	The merging operation over closed form systems
PI_1	The generator set of polynomial relations after the first iteration of the outer loop with $k \geq 1$ P-solvable inner loops
J_*	The ideal of polynomial invariants among the loop variables X with initial values X_0 of the P-solvable loop with conditional branches and assignments
IS_k	The set of possible iteration traces of the nested outer loop with $k \geq 1$ P-solvable inner loops

Index

Curriculum Vitæ

Personal data

Name	Laura Ildikó Kovács
Nationality	Hungarian
Citizenship	Romanian
Date and place of birth	April 26, 1980, Reşiţa, Romania

Contact

Email	kovacs@risc.uni-linz.ac.at
WWW	http://www.risc.uni-linz.ac.at/people/lkovacs

Education

1998	High school graduation at "Bartók Béla" Highschool Timişoara, Romania
1998–2002	Studies in mathematics and computer science at the West University of Timişoara, Romania
2002	Diploma degree in mathematics and computer science *Diploma thesis:* "Optical Music Recognition" *Thesis advisor:* Lucian Cucu
2002–2004 Feb.	MSc degree in computer science at the West University of Timişoara, Romania *Master thesis:* "Verification of Imperative Programs in Theorema" *Thesis advisors:* Prof. Tudor Jebelean (RISC-Linz) Prof. Viorel Negru (West University of Timişoara)
2003–2007	Doctorate studies in symbolic computation at the Research Institute for Symbolic Computation (RISC), Johannes Kepler University of Linz, Austria *Scientific advisors:* Prof. Tudor Jebelean (RISC-Linz) Prof. Andrei Voronkov (Univ. of Manchester)