RISC-Linz
Research Institute for Symbolic Computation
Johannes Kepler University
A-4040 Linz, Austria, Europe

Fifth International Symposium on

Symbolic Computation in Software Science

SCSS 2013

Symposium Proceedings

July 5-6, 2013
Research Institute for Symbolic Computation
Johannes-Kepler University Linz, Austria

Laura Kovács and Temur Kutsia
(Editors)

RISC-Linz Report Series No. 13-06

Supported by: the strategical program Innovative Upper Austria 2010 Plus and the Doctoral Program "Computational Mathematics" (W1214) - DK1 project.

Fifth International Symposium on

Symbolic Computation in Software Science

SCSS 2013

Symposium Proceedings

July 5-6, 2013
Research Institute for Symbolic Computation
Johannes-Kepler University Linz, Austria

Preface

Symbolic Computation is the science of computing with symbolic objects (terms, formulae, programs, representations of algebraic objects etc.). Powerful symbolic algorithms have been developed during the past decades like theorem proving, automated reasoning, software verification, model checking, rewriting, formalization of mathematics, network security, Gröbner bases, characteristic sets, telescoping for recurrence relations, etc.

The purpose of the International Symposium on Symbolic Computation in Software Science - SCSS is to promote research on theoretical and practical aspects of symbolic computation in software sciences. The symposium provides a forum for active dialog between researchers from several fields of computer algebra, algebraic geometry, algorithmic combinatorics, computational logic, and software analysis and verification.

This year's SCSS edition is the fifth in the SCSS workshop series and is organized at the Research Institute for Symbolic Computation (RISC) of the Johannes Kepler University Linz, during July 5-6, 2013. With the focus of promoting the exchange of ideas and experiences from applications of symbolic computation in software science, SCSS 2013 has attracted researchers from across a number of fields, including computer algebra, algebraic geometry, term rewriting, process algebras, and program verification.

We are very excited to have three excellent keynote speakers: Bruno Buchberger (RISC-Linz), Wei Li (Beihang University) and Joel Ouaknine (Oxford University). Our invited speakers have significant research expertise and insights into the challenges and opportunities for the growing SCSS community.

We would like to thank the program committee members and reviewers for all their efforts. Thanks also go to the steering committee members for their valuable advice and guidance. We would also like to thank Andrei Voronkov for his help on EasyChair. Finally, we also acknowledge partial funding from the strategical program Innovative Upper Austria 2010 Plus and the Doctoral Program "Computational Mathematics" (W1214) - DK1 project.

Laura Kovács, Programme Chair
Temur Kutsia, Symposium Chair

Symposium Organization

Symposium Chair

Temur Kutsia	Research Institute for Symbolic Computation - RISC, Austria

Program Chair

Laura Kovács	Chalmers University of Technology, Sweden

Program Committee

María Alpuente	Technical University of Valencia, Spain
Serge Autexier	DFKI, Germany
Nikolaj Bjorner	Microsoft Research, USA
Adel Bouhoula	Ecole superieure des communications de Tunis, Tunisia
Iliano Cervesato	Carnegie Mellon University - Qatar Campus, Qatar
Horatiu Cirstea	Loria, France
Jürgen Giesl	RWTH Aachen, Germany
Tetsuo Ida	Tsukuba University, Japan
Paul Jackson	School of Informatics, University of Edinburgh, UK
Tudor Jebelean	Research Institute for Symbolic Computation - RISC, Austria
Cezary Kaliszyk	University of Innsbruck, Austria
Fairouz Kamareddine	Heriot-Watt University, UK
Laura Kovács	Chalmers University of Technology, Sweden

Temur Kutsia	Research Institute for Symbolic Computation - RISC, Austria
Stephan Merz	INRIA-LORIA Nancy, France
Ali Mili	New Jersey Institute of Technology, USA
Yasuhiko Minamide	University of Tsukuba, Japan
Pierre-Etienne Moreau	INRIA-LORIA Nancy, France
André Platzer	Carnegie Mellon University, USA
Stefan Ratschan	Czech Academy of Sciences, The Czech Republic
Rachid Rebiha	University of Lugano, Switzerland and IC Unicamp, Brazil
Enric Rodríguez Carbonell	Technical University of Catalonia, Spain
Sorin Stratulat	Universite Paul Verlaine, France
Thomas Sturm	Max Planck Institute for Informatics, Germany

Additional Reviewers

Cabarcas, Daniel
Eder, Christian
Gay, Simon
Ghorbal, Khalil
Ghourabi, Fadoua
Kredel, Heinz
Martins, João G.
Vigneron, Laurent

Table of Contents

Mathematics of 21$^{\text{st}}$ Century: A Personal View

Bruno Buchberger

Research Institute for Symbolic Computation - RISC
Johannes Kepler University Linz, Austria
Bruno.Buchberger@risc.uni-linz.ac.at

Extended Abstract.

Mathematics of **19$^{\text{th}}$ century** and before was rich in content, intuition, strong ideas, motivation from natural sciences, and had a coherent view of all fields of mathematics (geometry, algebra, analysis, number theory, . . .).

Mathematics of **20$^{\text{th}}$ century** added three important aspects: mathematical **logic** as the mathematical meta-theory of mathematics; formalism and abstractness and reduction to first principles (**"Bourbakism"**); and the notion of **computing** (both as an exact mathematical notion and as the foundation of the revolutionary device "programmable computer"). In fact, the three aspects have a lot to do with each other and are fundamentally interrelated. However, in practice, the three aspects were (and still are) pursued by three different communities: The deep insights of logicians on the nature of mathematics are hardly used in ever-day practice of "working mathematicians"; "pure mathematicians" are often proud that they do not touch the computer except for writing e-mails, typing in LaTeX, and using web search; and computer scientists often think that what they are doing does not need mathematics or, even, is opposite to mathematics.

In my view, as indicated by trends in late 20$^{\text{th}}$ century, in **21$^{\text{st}}$ century** mathematics will evolve as - or return to be - a **unified body** of mathematical logic, abstract structural mathematics, and computer mathematics with no boundaries between the three aspects. "Working mathematicians" will have to master the three aspects equally well and integrate them in their daily work. More specifically, working in mathematics will proceed on the "object level" of developing new mathematical content (abstract knowledge and computational methods) and, at the same time, on the "meta-level" of developing automated reasoning methods for supporting research on the object level. This "massage of the mathematical brain" by jumping back and forth between the object and the meta-level will guide mathematics onto a new level of sophistication.

Symbolic computation is just a way of expressing this general view of mathematics of the 21$^{\text{st}}$ century and it also should be clear that **software science** is just another way of expressing the algorithmic aspect of this view.

In the talk, we will exemplify the spirit of this new type of mathematics by a report on the **Theorema** system being developed in the speaker's research group. Theorema is both a logic and a software frame for doing mathematics in the way sketched above. On the object level, Theorema allows to prove and program within the same logical frame and, on the meta-level, it allows to formulate reasoning techniques that help proving and programming on the object level. In particular, we will show how this type of doing mathematics allows to mimic the invention process behind the speaker's theory of Gröbner bases, which provides a general method for dealing with multivariate nonlinear polynomial systems.

A Semantic Framework for Program Debugging

Wei Li

State Key Laboratory of Software Development Environment
School of Computer Science and Engineering
Beihang University, China
liwei@nlsde.buaa.edu.cn

Abstract.

This work aims to build a semantic framework for automated debugging. A debugging process consists of tracing, locating, and fixing processes consecutively. The first two processes are accomplished by a tracing procedure and a locating procedure, respectively. The tracing procedure reproduces the execution of a failed test case with well-designed data structures and saves necessary information for locating bugs. The locating procedure will use the information obtained from the tracing procedure to locate ill-designed statements and to generate a fix-equation, the solution of which is a function that will be used to fix the bugs. A structural operational semantics is given to define the functions of the tracing and locating procedure. Both procedures are proved to terminate and produces one fix-equation. The main task of fixing process is to solve the fix-equation. It turns out that for a given failed test case, there exist three different types of solutions: 1. the bug is solvable, there exists a solution of the fix-equation, and the program can be repaired. 2. There exists a non-linear error in the program, the fix-equation generated at each round of the locating procedure is solvable, but a new bug will arise when the old bug is being fixed. 3. There exists a logical design error and the fix-equation is not solvable.

Decision Problems for Linear Recurrence Sequences

Joel Ouaknine

Department of Computer Science
Oxford University, United Kingdom
`Joel.Ouaknine@cs.ox.ac.uk`

Abstract.

Linear recurrence sequences (LRS), such as the Fibonacci numbers, permeate vast areas of mathematics and computer science. In this talk, we consider three natural decision problems for LRS, namely the *Skolem Problem* (does a given LRS have a zero?), the *Positivity Problem* (are all terms of a given LRS positive?), and the *Ultimate Positivity Problem* (are all but finitely many terms of a given LRS positive?). Such problems (and assorted variants) have applications in a wide array of scientific areas, such as theoretical biology (analysis of L-systems, population dynamics), economics (stability of supply-and-demand equilibria in cyclical markets, multiplier-accelerator models), software verification (termination of linear programs), probabilistic model checking (reachability and approximation in Markov chains, stochastic logics), quantum computing (threshold problems for quantum automata), discrete linear dynamical systems (reachability and invariance problems), as well as combinatorics, statistical physics, term rewriting, formal languages, cellular automata, generating functions, etc.

We shall see that these problems have deep and fascinating connections to rich mathematical concepts and conjectures, particularly in the fields of analytic and algebraic number theory, diophantine geometry and approximation, real algebraic geometry, mathematical logic, and complexity theory.

Parametric Exploration of Rewriting Logic Computations *

M. Alpuente[1], D. Ballis[2], F. Frechina[1] and J. Sapiña[1]

[1] DSIC-ELP, Universitat Politècnica de València,
Camino de Vera s/n, Apdo 22012, 46071 Valencia, Spain,
{alpuente,ffrechina,jsapina}@dsic.upv.es
[2] DIMI, Università degli Studi di Udine,
Via delle Scienze 206, 33100 Udine, Italy, demis.ballis@uniud.it

Abstract

This paper presents a parameterized technique for the inspection of Rewriting Logic computations that allows the non-deterministic execution of a given rewrite theory to be followed up in different ways. Starting from a selected state in the computation tree, the exploration is driven by a user-defined, inspection criterion that specifies the exploration mode. By selecting different inspection criteria, one can automatically derive useful debugging facilities such as program steppers and more sophisticated dynamic trace slicers that facilitate the detection of control and data dependencies across the computation tree. Our methodology, which is implemented in the Anima graphical tool, allows users to capture the impact of a given criterion, validate input data, and detect improper program behaviors.

1 Introduction

Program animation or *stepping* refers to the very common debugging technique of executing code one step at a time, allowing the user to inspect the program state and related data before and after the execution step. This allows the user to evaluate the effects of a given statement or instruction in isolation and thereby gain insight into the program behavior (or misbehavior). Nearly all modern IDEs, debuggers and testing tools currently support this mode of execution optionally, where animation is achieved either by forcing execution breakpoints, code instrumentation or instruction simulation.

Rewriting Logic (RWL) is a very general *logical* and *semantic framework*, which is particularly suitable for formalizing highly concurrent, complex systems (e.g., biological systems [7] and Web systems [2, 6]). RWL is efficiently implemented in the high-performance system Maude [9]. Roughly speaking, a *rewriting logic theory* seamlessly combines a *term rewriting system* (TRS), together with an *equational theory* that may include equations and axioms (i.e., algebraic laws such as commutativity, associativity, and unity) so that rewrite steps are performed *modulo* the equations and axioms.

In recent years, debugging and optimization techniques based on RWL have received growing attention [1, 13, 16, 17], but to the best of our knowledge, no practical animation tool for RWL/Maude has been formally developed. To debug Maude programs, Maude has a basic tracing facility that allows the user to advance through the program execution letting him/her select the statements to be traced, except for the application of algebraic axioms that are not

*This work has been partially supported by the EU (FEDER) and the Spanish MEC project ref. TIN2010-21062-C02-02, and by Generalitat Valenciana ref. PROMETEO2011/052. This work was carried out during the tenure of D. Ballis' ERCIM 'Alain Bensoussan" Postdoctoral Fellowship. The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n. 246016. F. Frechina was supported by FPU-ME grant AP2010-5681.

under user control and are never recorded as execution steps in the trace. All rewrite steps that are obtained by applying the equations or rules for the selected function symbols are shown in the output trace so that the only way to simplify the displayed view of the trace is by manually fixing the traceable equations or rules. Thus, the trace is typically huge and incomplete, and when the user detects an erroneous intermediate result, it is difficult to determine where the incorrect inference started. Moreover, this trace is either directly displayed or written to a file (in both cases, in plain text format) thus only being amenable for manual inspection by the user. This is in contrast with the enriched traces described below, which are complete (all execution steps are recorded by default) and can be sliced automatically so that they can be dramatically simplified in order to facilitate a specific analysis. Also, the trace can be directly displayed or delivered in its meta-level representation, which is very useful for further automated manipulation.

Contributions. This paper presents the first parametric (forward) exploration technique for RWL computations. Our technique is based on a generic animation algorithm that can be tuned to work with different modalities, including *incremental stepping* and *automated forward slicing.* The algorithm is fully general and can be applied for debugging as well as for optimizing any RWL-based tool that manipulates RWL computations. Our formulation takes into account the precise way in which Maude mechanizes the rewriting process and revisits all those rewrite steps in an instrumented, fine-grained way where each small step corresponds to the application of an equation, equational axiom, or rule. This allows us to explain the input execution trace with regard to the set of symbols of interest (input symbols) by tracing them along the execution trace so that, in the case of the forward slicing modality, all data that are not descendants of the observed symbols are simply discarded.

Related Work. Program animators have existed since the early years of programming. Although several steppers have been implemented in the functional programming community (see [10] for references), none of these systems applies to the animation and forward slicing of Maude computations. An algebraic stepper for Scheme is defined and formally proved in [10], which is included in the DrScheme programming environment. The stepper reduces Scheme programs to values (according to the reduction semantics of Scheme) and is useful for explaining the semantics of linguistic facilities and for studying the behavior of small programs. It explains a program's execution as a sequence of reduction steps based on the ordinary laws of algebra for the functional core of the language and more general algebraic laws for the rest of the language. In order to discover all of the steps that occur during the program evaluation, the stepper rewrites (or "instruments") the code. The inserted code uses a mechanism called "continuation marks" to store information about the program's execution as it is running and makes calls to the stepper before, after, and during the evaluation of each program expression. Continuation marks allow the stepper to reuse the underlying Scheme implementation without having to re-implement the evaluator. The stepper's implementation technique also applies to both ML and Haskell since it supports states, continuations, multiple threads of control, and lazy evaluation [10].

In [4, 5], an incremental, backward trace slicer was presented that generates a trace slice of an execution trace $\mathcal{T}$ by tracing back a set of symbols of interest along (an instrumented version of) $\mathcal{T}$, while data that are not required to produce the target symbols are simply discarded. This can be very helpful in debugging since any information that is not strictly needed to deliver a critical part of the result is discarded, which helps answer the question of what program components might affect a 'selected computation". However, for the dual problem of "what program components might be affected by a selected computation", a kind of forward expansion is needed which has been overlooked to date.

Plan of the paper. Section 2 recalls some fundamental notions of RWL, and Section 3 summarizes the rewriting modulo equational theories defined in Maude. Section 4 formulates the exploration as a parameterized procedure that is completely controlled by the user, while Section 5 formalizes three different inspection techniques that are obtained as an instance of the generic scheme. Finally, Section 6 reports on a prototypical implementation of the proposed techniques, and Section 7 concludes.

2 Preliminaries

Let us recall some important notions that are relevant to this work. We assume some basic knowledge of term rewriting [18] and Rewriting Logic [14]. Some familiarity with the Maude language [9] is also required.

We consider an *order-sorted signature* Σ, with a finite poset of sorts $(S, <)$ that models the usual subsort relation [9]. We assume an S-sorted family $\mathcal{V} = \{\mathcal{V}_s\}_{s \in S}$ of disjoint variable sets. $\tau(\Sigma, \mathcal{V})_s$ and $\tau(\Sigma)_s$ are the sets of terms and ground terms of sort s, respectively. We write $\tau(\Sigma, \mathcal{V})$ and $\tau(\Sigma)$ for the corresponding term algebras. The set of variables that occur in a term t is denoted by $\mathcal{V}ar(t)$. In order to simplify the presentation, we often disregard sorts when no confusion can arise.

A *position* w in a term t is represented by a sequence of natural numbers that addresses a subterm of t (Λ denotes the empty sequence, i.e., the root position). By notation $w_1.w_2$, we denote the concatenation of positions (sequences) w_1 and w_2. Positions are ordered by the prefix ordering; that is, given the positions w_1 and w_2, $w_1 \leq w_2$ if there exists a position u such that $w_1.u = w_2$.

Given a term t, we let $\mathcal{P}os(t)$ denote the set of positions of t. By $t|_w$, we denote the *subterm* of t at position w, and $t[s]_w$ specifies the result of *replacing the subterm* $t|_w$ by the term s.

A *substitution* $\sigma \equiv \{x_1/t_1, x_2/t_2, \ldots\}$ is a mapping from the set of variables $\mathcal{V}$ to the set of terms $\tau(\Sigma, \mathcal{V})$ which is equal to the identity almost everywhere except over a set of variables $\{x_1, \ldots, x_n\}$. The *domain* of σ is the set $Dom(\sigma) = \{x \in \mathcal{V} \mid x\sigma \neq x\}$. By *id* we denote the *identity* substitution. The application of a substitution σ to a term t, denoted $t\sigma$, is defined by induction on the structure of terms:

$$t\sigma = \begin{cases} x\sigma & \text{if } t = x, x \in \mathcal{V} \\ f(t_1\sigma, \ldots, t_n\sigma) & \text{if } t = f(t_1, \ldots, t_n), n \geq 0 \end{cases}$$

For any substitution σ and set of variables V, $\sigma_{\restriction V}$ denotes the substitution obtained from σ by restricting its domain to V (i.e., $\sigma_{\restriction V}(x) = x\sigma$ if $x \in V$, otherwise $\sigma_{\restriction V}(x) = x$). Given two terms s and t, a substitution σ is a *matcher* of t in s, if $s\sigma = t$. The term t is an *instance* of the term s, iff there exists a matcher σ of t in s. By $match_s(t)$, we denote the function that returns a matcher of t in s if such a matcher exists.

A (labelled) *equation* is an expression of the form $[l] : \lambda = \rho$, where $\lambda, \rho \in \tau(\Sigma, \mathcal{V})$, $Var(\rho) \subseteq Var(\lambda)$, and l is a label, i.e., a name that identifies the equation. A (labelled) *rewrite* rule is an expression of the form $[l] : \lambda \to \rho$, where $\lambda, \rho \in \tau(\Sigma, \mathcal{V})$, $Var(\rho) \subseteq Var(\lambda)$, and l is a label. When no confusion can arise, rule and equation labels are often omitted. The term λ (resp., ρ) is called *left-hand side* (resp. *right-hand side*) of the rule $\lambda \to \rho$ (resp. equation $\lambda = \rho$).

3 Rewriting Modulo Equational Theories

An *order-sorted equational theory* is a pair $E = (\Sigma, \Delta \cup B)$, where Σ is an order-sorted signature, Δ is a collection of (oriented) equations, and B is a collection of equational axioms (i.e., algebraic laws such as associativity, commutativity, and unity) that can be associated with any binary operator of Σ. The equational theory E induces a congruence relation on the term algebra $\tau(\Sigma, \mathcal{V})$, which is denoted by $=_E$. A *rewrite theory* is a triple $\mathcal{R} = (\Sigma, \Delta \cup B, R)$, where $(\Sigma, \Delta \cup B)$ is an order-sorted equational theory, and R is a set of rewrite rules.

Example 3.1

The following rewrite theory, encoded in Maude, specifies a close variant of the fault-tolerant client-server communication protocol of [15].

```
mod CLIENT-SERVER-TRANSF is inc INT + QID .
  sorts Content State Msg Cli Serv Addressee
    Sender Data CliName ServName Question Answer .
  subsort Msg Cli Serv < State .
  subsort Addressee Sender CliName ServName < Qid .
  subsort Question Answer Data < Nat .

  ops Srv Srv-A Srv-B Cli Cli-A Cli-B : -> Qid .
  op null : -> State .
  op _&_ : State State -> State [assoc comm id: null] .
  op _<-_ : Addressee Content -> Msg .
  op {_,_} : Sender Data -> Content .
  op [_,_,_,_] : CliName ServName Question Answer -> Cli .
  op na : -> Answer .
  op [_] : ServName -> Serv [ctor] .
  op f : ServName CliName Data -> Data .

  vars C S Addr : Qid .
  vars Q D A : Nat .
  var CNT : [Content] .

  eq [succ] : f(S, C, Q) = s(Q) .

  rl [req]   : [C, S, Q, na] =>
               [C, S, Q, na] & S <- {C, Q} .
  rl [reply] : S <- {C, Q} & [S] =>
               [S] & C <- {S, f(S, C, Q)} .
  rl [rec]   : C <- {S, D} & [C, S, Q, A] =>
               [C, S, Q, A] .
  rl [dupl]  : (Addr <- CNT) =>
               (Addr <- CNT) & (Addr <- CNT) .
  rl [loss]  : (Addr <- CNT) => null .
endm
```

The specification models an environment where several clients and servers coexist. Each server can serve many clients, while, for the sake of simplicity, we assume that each client communicates with a single server.

The names of clients and servers belong to the sort `Qid`. Clients are represented as 4-tuples of the form `[C, S, Q, D]`, where `C` is the client's name, `S` is the name of the server it wants to communicate with, `Q` is a natural number that identifies a client request, and `D` is either a natural number that represents the server response, or the constant value `na` (not available) when the response has not been yet received. Servers are stateless and are represented as structures `[S]`, with `S` being the server's name. All messages are represented as pairs of the form `Addr <- CNT`, where `Addr` is the addressee's name, and `CNT` stands for the message contents. Such contents are pairs {`Addr`,`N`}, with `Addr` being the sender's name and `N` being a number that represents either a request or a response.

The server `S` uses a function `f` (only known to the server itself) that, given a question `Q` from client `C`, the call `f(S,C,Q)` computes the answer `s(Q)` where `s(Q)` is the successor of `Q`. This function is specified by means of the equation `succ`.

Program states are formalized as a soup (multiset) of clients, servers, and messages, whereas the system behavior is formalized through five rewrite rules that model a faulty communication environment in which messages can arrive out of order, can be duplicated, and can be lost. Specifically, the rule `req` allows a client `C` to send a message with request `Q` to the server `S`. The rule `reply` lets the server `S` consume the client request `Q` and send a response message that is computed by means of the function `f`. The rule `rec` specifies the client reception of a server response `D` that should be stored in the client data structure. Indeed, the right-hand side [`C`, `S`, `Q`, `A`] of the rule includes an intentional, barely perceptible bug that does not let the client structure be correctly updated with the incoming response `D`. The correct right-hand side should be [`C`, `S`, `Q`, `D`]. Finally, the rules `dupl` and `loss` model the faulty environment and have the obvious meaning: messages can either be duplicated or lost.

Given a rewrite theory (Σ, E, R), with $E = \Delta \cup B$, the rewriting modulo E relation (in symbols, $\rightarrow_{R/E}$) can be defined by lifting the usual rewrite relation on terms [12] to the E-congruence classes $[t]_E$ on the term algebra $\tau(\Sigma, \mathcal{V})$ that are induced by $=_E$ [8]; that is, $[t]_E$ is the class of all terms that are equal to t *modulo* E. Unfortunately, $\rightarrow_{R/E}$ is in general undecidable since a rewrite step $t \rightarrow_{R/E} t'$ involves searching through the possibly infinite equivalence classes $[t]_E$ and $[t']_E$.

The exploration technique formalized in this work is formulated by considering the precise way in which Maude proves the rewrite steps (see Section 5.2 in [9]). Actually, the Maude interpreter implements rewriting modulo E by means of two much simpler relations, namely $\rightarrow_{\Delta,B}$ and $\rightarrow_{R,B}$. These allow rewrite rules and equations to be intermixed in the rewriting process by simply using an algorithm of matching modulo B. We define $\rightarrow_{R\cup\Delta,B}$ as $\rightarrow_{R,B}$ $\cup$ $\rightarrow_{\Delta,B}$. Roughly speaking, the relation $\rightarrow_{\Delta,B}$ uses the equations of Δ (oriented from left to right) as simplification rules: thus, for any term t, by repeatedly applying the equations as simplification rules, we eventually reach a normalized term $t{\downarrow_\Delta}$ to which no further equations can be applied. The term $t{\downarrow_\Delta}$ is called a *canonical form* of t w.r.t. Δ. On the other hand, the relation $\rightarrow_{R,B}$ implements rewriting with the rules of R, which might be non-terminating and non-confluent, whereas Δ is required to be terminating and Church-Rosser modulo B in order to guarantee the existence and unicity (modulo B) of a canonical form w.r.t. Δ for any term [9].

Formally, $\rightarrow_{R,B}$ and $\rightarrow_{\Delta,B}$ are defined as follows: given a rewrite rule $r = (\lambda \rightarrow \rho) \in R$ (resp., an equation $e = (\lambda = \rho) \in \Delta$), a substitution σ, a term t, and a position w of t, $t \overset{r,\sigma,w}{\rightarrow}_{R,B} t'$ (resp., $t \overset{e,\sigma,w}{\rightarrow}_{\Delta,B} t'$) iff $\lambda\sigma =_B t_{|w}$ and $t' = t[\rho\sigma]_w$. When no confusion can arise, we simply write $t \rightarrow_{R,B} t'$ (resp. $t \rightarrow_{\Delta,B} t'$) instead of $t \overset{r,\sigma,w}{\rightarrow}_{R,B} t'$ (resp. $t \overset{e,\sigma,w}{\rightarrow}_{\Delta,B} t'$).

Under appropriate conditions on the rewrite theory, a rewrite step modulo E on a term t can be implemented without loss of completeness by applying the following rewrite strategy [11]: (i) reduce t w.r.t. $\rightarrow_{\Delta,B}$ until the canonical form $t{\downarrow_\Delta}$ is reached; (ii) rewrite $t{\downarrow_\Delta}$ w.r.t. $\rightarrow_{R,B}$.

A *computation* $\mathcal{C}$ in the rewrite theory $(\Sigma, \Delta \cup B, R)$ is a rewrite sequence

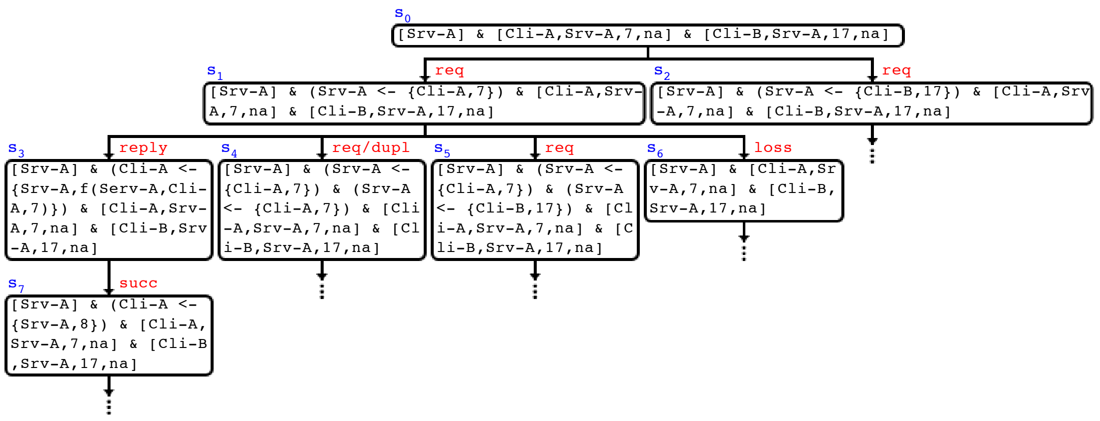
$$s_0 \rightarrow^*_{\Delta,B} s_0{\downarrow_\Delta} \rightarrow_{R,B} s_1 \rightarrow^*_{\Delta,B} s_1{\downarrow_\Delta} \ldots$$

that interleaves $\rightarrow_{\Delta,B}$ rewrite steps and $\rightarrow_{R,B}$ rewrite steps following the strategy mentioned above. Terms that appear in computations are also called (program) *states*.

A *computation tree* $\mathcal{T}_\mathcal{R}(s)$ for a term s and a rewrite theory $\mathcal{R}$ is a tree-like representation of all the possible computations in $\mathcal{R}$ that originate from the initial state s. More precisely, $\mathcal{T}_\mathcal{R}(s)$ is a labelled tree whose root node label identifies the initial state s and whose edges are labelled with rewrite steps of the form $t \rightarrow_{R\cup\Delta,B} t'$ so that any node in $\mathcal{T}_\mathcal{R}(s)$ represents a program state of a computation stemming from s.

Example 3.2

Consider the rewrite theory of Example 3.1 together with the initial state `[Srv-A] & [Cli-A, Srv-A,7,na] & [Cli-B,Srv-A,17,na]`. In this case, the computation tree consists of several infinite computations that start from the considered initial state and model interactions between clients `Cli-A` and `Cli-B` and server `Srv-A`. The computed tree is depicted in the following picture where, for the sake of simplicity, we only display the equations and rules that have been applied at each rewrite step, while other information such as the computed substitution and the rewrite position are omitted in the depicted tree.


s0
[Srv-A] & [Cli-A,Srv-A,7,na] & [Cli-B,Srv-A,17,na]
req
s1
[Srv-A] & (Srv-A <- {Cli-A,7}) & [Cli-A,Srv-A,7,na] & [Cli-B,Srv-A,17,na]
req
s2
[Srv-A] & (Srv-A <- {Cli-B,17}) & [Cli-A,Srv-A,7,na] & [Cli-B,Srv-A,17,na]
reply
s3
[Srv-A] & (Cli-A <- {Srv-A,f(Serv-A,Cli-A,7)}) & [Cli-A,Srv-A,7,na] & [Cli-B,Srv-A,17,na]
req/dupl
s4
[Srv-A] & (Srv-A <- {Cli-A,7}) & (Srv-A <- {Cli-A,7}) & [Cli-A,Srv-A,7,na] & [Cli-B,Srv-A,17,na]
req
s5
[Srv-A] & (Srv-A <- {Cli-A,7}) & (Srv-A <- {Cli-B,17}) & [Cli-A,Srv-A,7,na] & [Cli-B,Srv-A,17,na]
loss
s6
[Srv-A] & [Cli-A,Srv-A,7,na] & [Cli-B,Srv-A,17,na]
succ
s7
[Srv-A] & (Cli-A <- {Srv-A,8}) & [Cli-A,Srv-A,7,na] & [Cli-B,Srv-A,17,na]

Given a computation $\mathcal{C}$, it is always possible to expand $\mathcal{C}$ in an *instrumented* computation $\mathcal{C}_{inst}$ in which each application of the matching modulo B algorithm is mimicked by the explicit application of a suitable equational axiom, which is also oriented as a rewrite rule [3].

Also, typically hidden inside the B-matching algorithms, some flat/unflat transformations allow terms that contain operators that obey associative-commutative axioms to be rewritten. These transformations allow terms to be reordered and correctly parenthesized in order to enable subsequent rewrite steps. Basically, this is achieved by producing a single, auxiliary representative of their AC congruence class [3]. For example, consider a binary AC operator f together with the standard lexicographic ordering over symbols. Given the B-equivalence $f(b,f(f(b,a),c)) =_B f(f(b,c),f(a,b))$, we can represent it by using the "internal sequence" of transformations $f(b,f(f(b,a),c)) \xrightarrow{flat}{}^*_B f(a,b,b,c) \xrightarrow{unflat}{}^*_B f(f(b,c),f(a,b))$, where the first subsequence corresponds to a *flattening* transformation sequence that obtains the AC canonical form, while the second one corresponds to the inverse, unflattening transformation. This way, any given instrumented computation consists of a sequence of (standard) rewrites using the equations ($\to_\Delta$), rewrite rules ($\to_R$), equational axioms and flat/unflat transformations ($\to_B$). By abuse of notation, since equations and axioms are both interpreted as rewrite rules in our formulation, we often abuse the notation $\lambda \to \rho$ to denote rules as well as (oriented) equations and axioms. Given an instrumented computation $\mathcal{C}_{inst}$, by $|\mathcal{C}_{inst}|$ we denote its *length* —that is, the number of rewrite steps that occur in $\mathcal{C}_{inst}$. We use the functions $head(\mathcal{C}_{inst})$ and $tail(\mathcal{C}_{inst})$ to respectively select the first rewrite step in $\mathcal{C}_{inst}$ and the instrumented computation yielded by removing the first rewrite step from $\mathcal{C}_{inst}$. In the sequel, we also assume the existence of a function $instrument(\mathcal{C})$, which takes a computation $\mathcal{C}$ and delivers its instrumented counterpart.

Example 3.3

Consider the rewrite theory of Example 3.1 together with the following computation:

$$\begin{array}{rl} \mathcal{C} = & \texttt{[Cli, Srv, 7, na] \& [Srv] \& Cli <- \{Srv, f(Srv, Cli, 7)\}} \to_{\Delta,B} \\ & \texttt{[Cli, Srv, 7, na]\& [Srv] \& Cli <- \{Srv, 8\}} \to_{R,B} \\ & \texttt{[Cli, Srv, 7 , 7] \& [Srv]} \end{array}$$

The corresponding instrumented computation $\mathcal{C}_{inst}$, which is produced by $instrument(\mathcal{C})$, is given by 1) suitably parenthesizing states by means of flattening/unflattening transformations when needed, and 2) making commutative "steps" explicit by using the (oriented) equational axiom (`X & Y` $\rightarrow$ `Y & X`) in B that models the commutativity property of the (juxtaposition) operator &. Note these transformations are needed to enable the application of the rule `rec` (of R) to the seventh state.

$$
\begin{array}{ll}
\mathcal{C}_{inst} = & \texttt{[Cli, Srv, 7, na] \& [Srv] \& Cli <- \{Srv, f(Srv, Cli, 7)\}} \xrightarrow{succ}_{\Delta} \\
& \texttt{[Cli, Srv, 7, na] \& [Srv] \& Cli <- \{Srv, 8\}} \xrightarrow{unflat}_{B} \\
& \texttt{[Cli, Srv, 7, na] \& ([Srv] \& Cli <- \{Srv, 8\})} \xrightarrow{comm}_{B} \\
& \texttt{[Cli, Srv, 7, na] \& (Cli <- \{Srv, 8\} \& [Srv])} \xrightarrow{flat}_{B} \\
& \texttt{[Cli, Srv, 7, na] \& Cli <- \{Srv, 8\} \& [Srv]} \xrightarrow{unflat}_{B} \\
& \texttt{([Cli, Srv, 7, na] \& Cli <- \{Srv, 8\}) \& [Srv]} \xrightarrow{comm}_{B} \\
& \texttt{(Cli <- \{Srv, 8\} \& [Cli, Srv, 7, na]) \& [Srv]} \xrightarrow{rec}_{R} \\
& \texttt{[Cli, Srv, 7, 7] \& [Srv]}
\end{array}
$$

4 Exploring the Computation Tree

Computation trees are typically large and complex objects to deal with because of the highly-concurrent, nondeterministic nature of rewrite theories. Also, their complete generation and inspection are generally not feasible since some of their branches may be infinite as they encode nonterminating computations.

However, one may still be interested in analysing a fragment of a given computation tree for debugging or program comprehension purposes. This section presents an exploration technique that allows the user to incrementally generate and inspect a portion $\mathcal{T}^{\bullet}_{\mathcal{R}}(s^{\bullet})$ of a computation tree $\mathcal{T}_{\mathcal{R}}(s)$ by expanding (fragments of) its program states into their descendants starting from the root node. The exploration is an interactive procedure that is completely controlled by the user, who is free to choose the program state fragments to be expanded.

4.1 Expanding a Program State

A state *fragment* of a state s is a term $s^{\bullet}$ that hides part of the information in s, that is, the irrelevant data in s are simply replaced by special $\bullet$-variables of appropriate sort, denoted by $\bullet_i$, with $i = 0, 1, 2, \ldots$. Given a state fragment $s^{\bullet}$, a *meaningful* position p of $s^{\bullet}$ is a position $p \in \mathcal{P}os(s^{\bullet})$ such that $s^{\bullet}_{|p} \neq \bullet_i$, for all $i = 0, 1, \ldots$. By $\mathcal{MP}os(s^{\bullet})$, we denote the set that contains all the meaningful positions of $s^{\bullet}$. Symbols that occur at meaningful positions of a state fragment are called *meaningful* symbols. By $\mathcal{V}ar^{\bullet}(exp)$ we define the set of all $\bullet$-variables that occur in the expression *exp*.

Basically, a state fragment records just the information the user wants to observe of a given program state.

The next auxiliary definition formalizes the function $fragment(t, P)$ that allows a state fragment of t to be constructed w.r.t. a set of positions P of t. The function *fragment* relies on the function $fresh^{\bullet}$ whose invocation returns a (fresh) variable $\bullet_i$ of appropriate sort, which is distinct from any previously generated variable $\bullet_j$.

Definition 4.1 *Let $t \in \tau(\Sigma, \mathcal{V})$ be a state, and let P be a set of positions s.t. $P \subseteq \mathcal{P}os(t)$. Then, the function $fragment(t, P)$ is defined as follows.*

$$fragment(t, P) = recfrag(t, P, \Lambda), \ \textit{where}$$

```
function inspect($s^\bullet$, $\mathcal{C}_{instr}$, $\mathcal{I}$)
1. if $|\mathcal{C}_{instr}| = 1$ then
2.   return $\mathcal{I}(s^\bullet, \mathcal{C}_{instr})$
3. elseif $|\mathcal{C}_{instr}| > 1$
4.   if $\mathcal{I}(s^\bullet, head(\mathcal{C}_{instr}))$ != nil then
5.     return $inspect(\mathcal{I}(s^\bullet, head(\mathcal{C}_{instr})),$
                      $tail(\mathcal{C}_{instr}), \mathcal{I})$
6.   else
7.     return nil
8.   end
9. end
endf
```

Figure 1: The *inspect* function.

```
function expand($s^\bullet$, $s$, $\mathcal{R}$, $\mathcal{I}$)
1. $\mathcal{E}^\bullet_{\mathcal{R}} = \emptyset$
2. for each $s \overset{r,\sigma,w}{\rightarrow}_{R\cup\Delta,B} t$
              with $w \in \mathcal{MP}os(s^\bullet)$
3.   $\mathcal{C}_{inst} = instrument(s \overset{r,\sigma,w}{\rightarrow}_{R\cup\Delta,B} t)$
4.   $t^\bullet = inspect(s^\bullet, \mathcal{C}_{inst}, \mathcal{I})$
5.   if $t^\bullet \neq$ nil then $\mathcal{E}^\bullet_{\mathcal{R}} = \mathcal{E}^\bullet_{\mathcal{R}} \cup \{s^\bullet \overset{r}{\rightarrow} t^\bullet\}$
6. end
7. return $\mathcal{E}^\bullet_{\mathcal{R}}$
endf
```

Figure 2: The *expand* function.

$$recfrag(t,P,p) = \begin{cases} f(recfrag(t_1,P,p.1),\ldots,recfrag(t_n,P,p.n)) & \\ & \textit{if } t{=}f(t_1,\ldots,t_n) \textit{ and } p \in \bar{P} \\ t & \textit{if } t \in \mathcal{V} \textit{ and } p \in \bar{P} \\ fresh^\bullet & \textit{otherwise} \end{cases}$$

and $\bar{P} = \{u \mid u \leq p \wedge p \in P\}$ *is the* prefix closure *of* P.

Roughly speaking, *fragment*(t, P) yields a state fragment of t w.r.t. a set of positions P that includes all symbols of t that occur within the paths from the root to any position in P, while each maximal subterm $t_{|p}$, with $p \notin P$, is replaced by means of a freshly generated $\bullet$-variable.

Example 4.2

Let $t = d(f(g(a, h(b)), c), a)$ be a state, and let $P = \{1.1,\ 1.2\}$ be a set of positions of t. By applying Definition 4.1, we get the state fragment $t^\bullet = \textit{fragment}(t, P) = d(f(g(\bullet_1, \bullet_2), c), \bullet_3)$ and the set of meaningful positions $\mathcal{MP}os(t^\bullet) = \{\Lambda, 1, 1.1, 1.2\}$.

An *inspection criterion* is a function $\mathcal{I}(s^\bullet, s \overset{r,\sigma,w}{\rightarrow}_K t)$ that, given a rewrite step $s \overset{r,\sigma,w}{\rightarrow}_K t$, with $K \in \{\Delta, R, B\}$ and a state fragment $s^\bullet$ of s, computes a state fragment $t^\bullet$ of the state t. Roughly speaking, an inspection criterion controls the information content conveyed by term fragments resulting from the execution of standard rewrite steps. Hence, distinct implementations of the inspection criteria $\mathcal{I}(s^\bullet, s \overset{r,\sigma,w}{\rightarrow}_K t)$ may produce distinct fragments $s^\bullet \overset{r}{\rightarrow} t^\bullet$ of the considered rewrite step. We assume that the special value **nil** is returned by the inspection criterion, whenever no fragment $t^\bullet$ can be delivered. Several examples of inspection criteria are shown in Section 5.

The function *inspect* of Figure 1 allows an inspection criterion $\mathcal{I}$ to be sequentially applied along an instrumented computation $\mathcal{C}_{instr}$ in order to generate the fragment of the last state of the computation. Specifically, given an instrumented computation $s_0 \rightarrow_K s_1 \rightarrow_K \ldots \rightarrow s_n$, $n > 0$, the computation is traversed and the inspection criterion $\mathcal{I}$ is recursively applied on each rewrite step $s_i \rightarrow_K s_{i+1}$ w.r.t. the input fragment $s^\bullet_i$ to generate the next fragment $s^\bullet_{i+1}$.

The expansion of a single program state is specified by the function *expand*$(s^\bullet, s, \mathcal{R}, \mathcal{I})$ of Figure 2, which takes as input a state s and its fragment $s^\bullet$ to be expanded w.r.t. a rewrite theory $\mathcal{R}$ and an inspection criterion $\mathcal{I}$. Basically, *expand* unfolds the state fragment $s^\bullet$ w.r.t. all the possible rewrite steps $s \overset{r,\sigma,w}{\rightarrow}_{R\cup\Delta,B} t$ that occur at the meaningful positions of $s^\bullet$ and stores the corresponding fragments of $s^\bullet \overset{r}{\rightarrow} t^\bullet$ in the set $\mathcal{E}^\bullet$. Note that, to compute the state fragment $t^\bullet$ for a rewrite step $s \overset{r,\sigma,w}{\rightarrow}_{R\cup\Delta,B} t$ and a state fragment $s^\bullet$, *expand* first generates the instrumented computation $\mathcal{C}_{inst}$ of the considered step and then applies the inspection criterion $\mathcal{I}$ over $\mathcal{C}$ by using the *inspect* function.

4.2 Computing a Fragment of the Computation Tree

The construction of a fragment $\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0)$ of a computation tree $\mathcal{T}_\mathcal{R}(s_0)$ is specified by the function *explore* given in Figure 3. Essentially, *explore* formalizes an interactive procedure that starts from a tree fragment (built using the auxiliary function *createTree*), which only consists of the root node $s^\bullet_0$, and repeatedly uses the function *expand* to compute rewrite step fragments that correspond to the visited tree edges, w.r.t. a given inspection criterion. The tree $\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0)$ is built by choosing, at each loop iteration of the algorithm, the tree node that represents the state fragment to be expanded by means of the auxiliary function *pickLeaf*$(\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0))$, which allows the user to freely select a node $s^\bullet$ from the frontier of the current tree $\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0)$. Then, $\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0)$ is augmented by calling *addChildren*$(\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0), s^\bullet, expand(s^\bullet, s, \mathcal{R}, \mathcal{I}))$. This function call adds all the edges $s^\bullet \rightarrow t^\bullet$, which are obtained by expanding $s^\bullet$, to the tree $\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0)$.

The special value **EoE** (End of Exploration) is used to terminate the inspection process: when the function *pickLeaf*$(\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0))$ is equal to **EoE**, no state to be expanded is selected and the exploration terminates delivering the computed fragment $\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0)$.

```
function $explore(s^\bullet_0, s_0, \mathcal{R}, \mathcal{I})$
1. $\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0) = createTree(s^\bullet_0)$
2. while($s^\bullet = pickLeaf(\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0)) \neq$ EoE) do
3.    $\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0) = addChildren(\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0), s^\bullet, expand(s^\bullet, s, \mathcal{R}, \mathcal{I}))$
4. od
5. return $\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0)$
endf
```

Figure 3: The *explore* function.

5 Particularizing the Exploration

The methodology given in Section 4 provides a generic scheme for the exploration of computation trees w.r.t. a given inspection criterion $\mathcal{I}$ that must be provided by the user. In this section, we show three implementations of the criterion $\mathcal{I}$ that produce three distinct exploration strategies. In the first case, the considered criterion allows an interactive program stepper to be derived in which rewriting logic computations of interest can be animated. In the second case, we implement a partial stepper that allows computations with partial inputs to be animated. Finally, in the last example, the chosen inspection criterion implements an automated, forward slicing technique that allows relevant control and data information to be extracted from computation trees.

5.1 Interactive Stepper

Given a computation tree $\mathcal{T}_\mathcal{R}(s_0)$ for an initial state s_0 and a rewrite theory $\mathcal{R}$, the stepwise inspection of the computation tree can be directly implemented by instantiating the exploration scheme of Section 4 with the inspection criterion

$$\mathcal{I}_{step}(s^\bullet, s \overset{r,\sigma,w}{\rightarrow}_K t) = t$$

that always returns the reduced state t of the rewrite step $s \overset{r,\sigma,w}{\to}_K t$, with $K \in \{\Delta, R, B\}$.

This way, by starting the exploration from a state fragment that corresponds to the whole initial state s_0 (i.e., $s_0^\bullet = s_0$), the call $\mathit{explore}(s_0, \mathcal{R}, \mathcal{I}_{step})$ generates a fragment $\mathcal{T}_\mathcal{R}^\bullet(s_0^\bullet)$ of the computation tree $\mathcal{T}_\mathcal{R}(s_0)$ whose topology depends on the program states that the user decides to expand during the exploration process.

Example 5.1

Consider the rewrite theory $\mathcal{R}$ in Example 3.1 and the computation tree in Example 3.2. Assume the user starts the exploration by calling $\mathit{explore}(s_0, \mathcal{R}, \mathcal{I}_{step})$, which allows the first level of the computation tree to be unfolded by expanding the initial state s_0 w.r.t. the inspection criterion $\mathcal{I}_{step}$. This generates the tree $\mathcal{T}_\mathcal{R}^\bullet(s_0)$ containing the edges $\{s_0 \overset{req}{\to} s_1, s_0 \overset{req}{\to} s_2\}$. Now, if the user carries on with the exploration of program state s_1 and then quits, s_1 will be expanded and the tree $\mathcal{T}_\mathcal{R}^\bullet(s_0)$ will be augmented accordingly. Specifically, the resulting $\mathcal{T}_\mathcal{R}^\bullet(s_0)$ will include the following edges $\{s_0 \overset{req}{\to} s_1, s_0 \overset{req}{\to} s_2, s_1 \overset{succ}{\to} s_3, s_1 \overset{req}{\to} s_4, s_1 \overset{dupl}{\to} s_4, s_1 \overset{req}{\to} s_5, s_1 \overset{loss}{\to} s_6\}$.

It is worth noting that all the program state fragments produced by the program stepper defined above are "concrete" (i.e. state fragments that do not include $\bullet$-variables). However, sometimes it may be useful to work with partial information and hence with state fragments that abstract "concrete" program states by using $\bullet$-variables. This approach may help to focus user's attention on the parts of the program states that the user wants to observe, disregarding unwanted information and useless rewrite steps.

Example 5.2

Consider the following two rewrite rules $[r_1] : f(x, b) \to g(x)$ and $[r_2] : f(a, y) \to h(y)$ together with the initial state $f(a, b)$. Then, the computation tree in this case is finite and only contains the tree edges $f(a, b) \overset{r_1,\{x/a\},\Lambda}{\to} g(a)$ and $f(a, b) \overset{r_2,\{y/b\},\Lambda}{\to} h(b)$. Now, consider the state fragment $f(\bullet_1, b)$, where only the second input argument is relevant. If we decided to expand the initial state fragment $f(\bullet_1, b)$, we would get the tree fragment represented by the single rewrite step $f(\bullet_1, b) \overset{r_1}{\to} g(\bullet_1)$, since the partial input encoded into $f(\bullet_1, b)$ cannot be rewritten via r_2.

In light of Example 5.2 and our previous considerations, we define the following inspection criterion

$$\mathcal{I}_{pstep}(s^\bullet, s \overset{r,\sigma,w}{\to}_K t) = \textbf{if } s^\bullet \overset{r,\sigma^\bullet,w}{\to}_K t^\bullet \textbf{ then return } t^\bullet \textbf{ else return nil}$$

Roughly speaking, given a rewrite step $\mu : s \overset{r,\sigma,w}{\to}_K t$, with $K \in \{\Delta, R, B\}$, the criterion $\mathcal{I}_{pstep}$ returns a state fragment $t^\bullet$ of the reduced state t, whenever $s^\bullet$ can be rewritten to $t^\bullet$ using the very same rule r and position w that occur in μ.

The particularization of the exploration scheme with the criterion $\mathcal{I}_{pstep}$ allows an interactive, partial stepper to be derived, in which the user can work with state information of interest, thereby producing more compact and focused representations of the visited fragments of the computation trees.

5.2 Forward Trace Slicer

Forward trace slicing is a program analysis technique that allows computations to be simplified w.r.t. a selected fragment of their initial state. More precisely, given a computation $\mathcal{C}$ with initial state s_0 and a state fragment $s_0^\bullet$ of s_0, forward slicing yields a simplified view $\mathcal{C}^\bullet$ of $\mathcal{C}$ in which each state s of the original computation is replaced by a state fragment $s^\bullet$ that only

```
**function** $\mathcal{I}_{slice}(s^\bullet, s \overset{\lambda\to\rho,\sigma,w}{\to} t)$
1. **if** $w \in \mathcal{MP}os(s^\bullet)$ **then**
2.     $\theta = \{x/fresh^\bullet \mid x \in Var(\lambda)\}$
3.     $\lambda^\bullet = fragment(\lambda, \mathcal{MP}os(\mathcal{V}ar^\bullet(s^\bullet_{|w})) \cap \mathcal{P}os(\lambda))$
4.     $\psi_\lambda = \langle\!|\theta, match_{\lambda^\bullet}(s^\bullet_{|w})|\!\rangle$
5.     $t^\bullet = s^\bullet[\rho\psi_\lambda]_w$
6. **else**
7.     $t^\bullet =$ **nil**
8. **fi**
9. **return** $t^\bullet$
**endf**
```

Figure 4: Inspection criterion that models forward slicing of a rewrite step

records the information that depends on the meaningful symbols of $s^\bullet_0$, while unrelated data are simply pruned away.

In the following, we define an inspection criterion $\mathcal{I}_{slice}$ that implements the forward slicing of a single rewrite step. The considered criterion takes two parameters as input, namely, a rewrite step $\mu = (s \overset{r,\sigma,w}{\to}_K t)$ (with $r = \lambda \to \rho$ and $K \in \{\Delta, R, B\}$) and a state fragment $s^\bullet$ of a state s. It delivers the state fragment $t^\bullet$ which includes only those data that are related to the meaningful symbols of $s^\bullet$. Intuitively, the state fragment $t^\bullet$ is obtained from $s^\bullet$ by "rewriting" $s^\bullet$ at position w with the rule r and a suitable substitution that abstracts unwanted information of the computed substitution with $\bullet$-variables. A rigorous formalization of the inspection criterion $\mathcal{I}_{slice}$ is provided by the algorithm in Figure 4.

Note that, by adopting the inspection criterion $\mathcal{I}_{slice}$, the exploration scheme of Section 4 automatically turns into an interactive, forward trace slicer that expands program states using the slicing methodology encoded into the inspection criterion $\mathcal{I}_{slice}$. In other words, given a computation tree $\mathcal{T}_\mathcal{R}(s_0)$ and a user-defined state fragment $s^\bullet_0$ of the initial state s_0, any branch $s^\bullet_0 \to s^\bullet_1 \ldots \to s^\bullet_n$ in the tree $\mathcal{T}^\bullet_\mathcal{R}(s^\bullet_0)$, which is computed by the *explore* function, is the sliced counterpart of a computation $s_0 \to_{R\cup\Delta,B} s_1 \ldots \to_{R\cup\Delta,B} s_n$ (w.r.t. the state fragment $s^\bullet_0$) that appears in the computation tree $\mathcal{T}_\mathcal{R}(s_0)$.

Roughly speaking, the inspection criterion $\mathcal{I}_{slice}$ works as follows. When the rewrite step μ occurs at a position w that is not a meaningful position of $s^\bullet$ (in symbols, $w \notin \mathcal{MP}os(s^\bullet)$), trivially μ does not contribute to producing the meaningful symbols of $t^\bullet$. This amounts to saying that no relevant information descends from the state fragment $s^\bullet$ and, hence, the function returns the **nil** value.

On the other hand, when $w \in \mathcal{MP}os(s^\bullet)$, the computation of $t^\bullet$ involves a more in-depth analysis of the rewrite step, which is based on a refinement process that allows the descendants of $s^\bullet$ in $t^\bullet$ to be computed. The following definition is auxiliary.

Definition 5.3 (substitution update) *Let σ_1 and σ_2 be two substitutions. The* update *of σ_1 w.r.t. σ_2 is defined by the operator $\langle\!|\_,\_|\!\rangle$ as follows:*
$\langle\!|\sigma_1, \sigma_2|\!\rangle = \sigma_{\restriction Dom(\sigma_1)}$, *where*

$$x\sigma = \begin{cases} x\sigma_2 & \textit{if } x \in Dom(\sigma_1) \cap Dom(\sigma_2) \\ x\sigma_1 & \textit{otherwise} \end{cases}$$

The operator $\langle\!|\sigma_1, \sigma_2|\!\rangle$ updates (overrides) a substitution σ_1 with a substitution σ_2, where both σ_1 and σ_2 may contain $\bullet$-variables. The main idea behind $\langle\!|\_,\_|\!\rangle$ is that, for the slicing of the rewrite step μ, all variables in the applied rewrite rule r are naïvely assumed to be initially bound to irrelevant data $\bullet$, and the bindings are incrementally updated as we apply the rule r.

More specifically, we initially define the substitution $\theta = \{x/fresh^\bullet \mid x \in Var(\rho)\}$ that binds each variable in $\lambda \to \rho$ to a fresh $\bullet$-variable. This corresponds to assuming that all the information in μ, which is introduced by the substitution σ, can be marked as irrelevant. Then, θ is refined as follows.

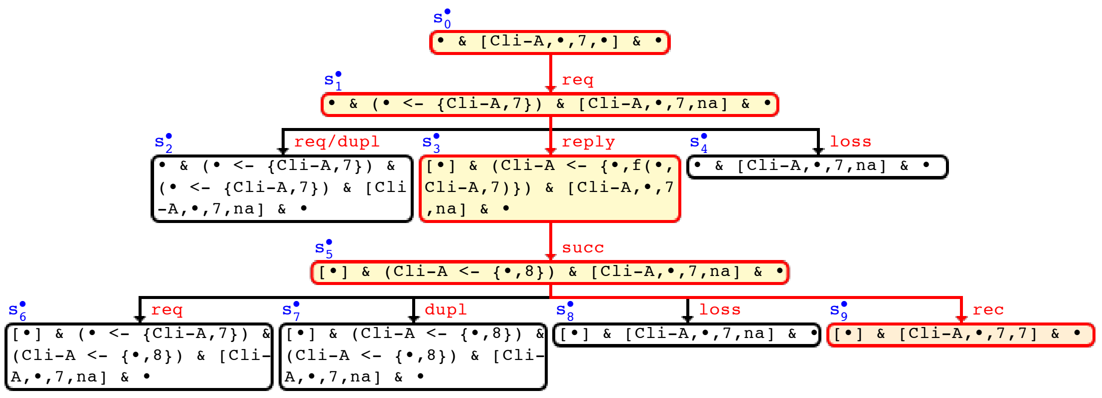
We first compute the state fragment $\lambda^\bullet = fragment(\lambda, \mathcal{MP}os(\mathcal{V}ar^\bullet(s^\bullet_{|w})) \cap \mathcal{P}os(\lambda))$ that only records the meaningful symbols of the left-hand side λ of the rule r w.r.t. the set of meaningful positions of $s^\bullet_{|w}$. Then, by matching $\lambda^\bullet$ with $s^\bullet_{|w}$, we generate a matcher $match_{\lambda^\bullet}(s^\bullet_{|w})$ that extracts the meaningful symbols from $s^\bullet_{|w}$. Such a matcher is then used to compute ψ_λ, which is an update of θ w.r.t. $match_{\lambda^\bullet}(s^\bullet_{|w})$ containing the meaningful information to be tracked. Finally, the state fragment $t^\bullet$ is computed from $s^\bullet$ by replacing its subterm at position w with the instance $\rho\psi_\lambda$ of the right-hand side of the applied rule r. This way, we can transfer all the relevant information marked in $s^\bullet$ into the fragment of the resulting state $t^\bullet$.

Example 5.4

Consider the computation tree of Example 3.2 whose initial state is

$$s_0 = \texttt{[Srv-A] \& [Cli-A,Srv-A,7,na] \& [Cli-B,Srv-A,17,na]}.$$

Let $s^\bullet_0 = \bullet$ `&` `[Cli-A,`$\bullet$`,7,`$\bullet$`]` `&` $\bullet$ be a state fragment[1] of s_0 where only request `7` of Client `Cli-A` is considered of interest. By sequentially expanding the nodes $s^\bullet_0$, $s^\bullet_1$, $s^\bullet_3$, and $s^\bullet_5$ w.r.t. the inspection criterion $\mathcal{I}_{slice}$, we get the following fragment of the given computation tree:


s0•
• & [Cli-A,•,7,•] & •
req
s1•
• & (• <- {Cli-A,7}) & [Cli-A,•,7,na] & •
req/dupl
s2•
• & (• <- {Cli-A,7}) & (• <- {Cli-A,7}) & [Cli-A,•,7,na] & •
reply
s3•
[•] & (Cli-A <- {•,f(•,Cli-A,7)}) & [Cli-A,•,7,na] & •
loss
s4•
• & [Cli-A,•,7,na] & •
succ
s5•
[•] & (Cli-A <- {•,8}) & [Cli-A,•,7,na] & •
req
s6•
[•] & (• <- {Cli-A,7}) & (Cli-A <- {•,8}) & [Cli-A,•,7,na] & •
dupl
s7•
[•] & (Cli-A <- {•,8}) & (Cli-A <- {•,8}) & [Cli-A,•,7,na] & •
loss
s8•
[•] & [Cli-A,•,7,na] & •
rec
s9•
[•] & [Cli-A,•,7,7] & •

Note that the slicing process automatically computes a tree fragment that represents a partial view of the protocol interactions from client `Cli-A`'s perspective. Actually, irrelevant information is hidden and rules applied on irrelevant positions are directly ignored, which allows a simplified fragment to be obtained favoring its inspection for debugging and analysis purposes. In fact, if we observe the highlighted computation in the tree, we can easily detect the wrong behaviour of the rule `rec`. Specifically, by inspecting the state fragment $s^\bullet_9 = ([\bullet]$ `&` $[\texttt{Cli}-\texttt{A}, \bullet, 7, 7]$ `&` $\bullet)$, which is generated by an application of the rule `rec`, we immediately realize that the response `8` produced in the parent state $s^\bullet_5$ has not been stored in $s^\bullet_9$, which clearly indicates a buggy implementation of the considered rule.

[1] Throughout the example, we omit indices from the considered $\bullet$-variables to keep notation lighter and improve readability. So, any variable $\bullet_i$, $i = 0, 1, \ldots$, is simply denoted by $\bullet$.

Finally, it is worth noting that the forward trace slicer implemented via the criterion $\mathcal{I}_{slice}$ differs from the partial stepper given at the end of Section 5.1. Given a state fragment $s^\bullet$ and a rewrite step $\overset{r,\sigma,w}{\to}_K t$, $\mathcal{I}_{slice}$ always yields a fragment $t^\bullet$ when the rewrite occurs at a meaningful position. By contrast, the inspection criterion $\mathcal{I}_{pstep}$ encoded in the partial stepper may fail to provide a computed fragment $t^\bullet$ when $s^\bullet$ does not rewrite to $t^\bullet$.

Example 5.5

Consider the same rewrite rules and initial state $f(\bullet_1, b)$ of Example 5.2. By expanding $f(\bullet_1, b)$ w.r.t. the inspection criterion $\mathcal{I}_{slice}$, we get the computation tree fragment with tree edges $f(\bullet_1, b) \overset{r_1}{\to} g(\bullet_1)$ and $f(\bullet_1, b) \overset{r_2}{\to} h(b)$, whereas the partial stepper only computes the tree edge $f(\bullet_1, b) \overset{r_1}{\to} g(\bullet_1)$ as shown in Example 5.2.

6 Implementation

The exploration methodology developed in this paper has been implemented in the Anima tool, which is publicly available at `http://safe-tools.dsic.upv.es/anima/`. The underlying rewriting machinery of Anima is written in Maude and consists of about 150 Maude function definitions (approximately 1600 lines of source code).

Anima also comes with an intuitive Web user interface based on AJAX technology, which allows users to graphically display and animate computation tree fragments. The core exploration engine is specified as a RESTful Web service by means of the Jersey JAX-RS API.

The architecture of Anima is depicted in Figure 5 and consists of five main components: Anima Client, JAX-RS API, Anima Web Service, Database, and Anima Core. The Anima Client is purely implemented in HTML5 and JSP. It represents the front-end layer of our tool and provides an intuitive, versatile Web user interface, which interacts with the Anima Web Service to invoke the capabilities of the Anima Core and save partial results in the Database component.

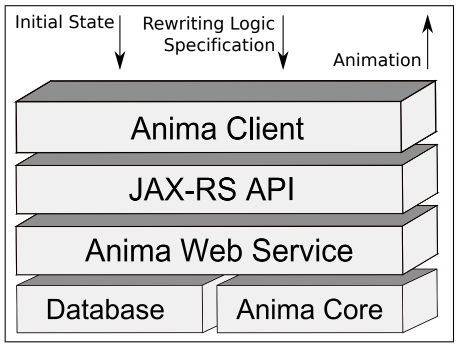


Figure 5: Anima architecture.

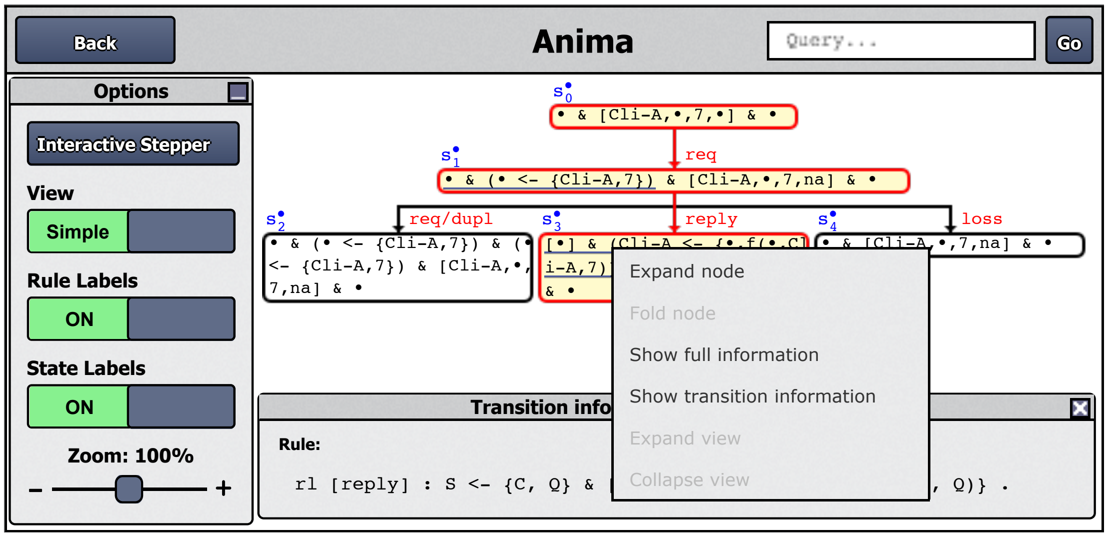
A screenshot that shows the Anima tool at work on the case study that is described in Example 5.4 is given in Figure 6.

These are the main features provided by Anima:

1. *Inspection strategies.* The tool implements the three inspection strategies described in Section 5. As shown in Figure 6, the user can select a strategy by using the selector provided in the option pane.

2. *Expanding/Folding program states.* The user can expand or fold states by right-clicking on them with the mouse and by selecting the *Expand/Fold Node* options from the contextual menu. For instance, in Figure 6, a state fragment on the frontier of the computed tree has been selected and is ready to be expanded through the *Expand Node* option.

3. *Selecting meaningful symbols.* State fragments can be specified by highlighting the state symbols of interest either directly on the tree or in the detailed information window.



Rule:

```
rl [reply] : S <- {C, Q} & [ , Q)} .
```

Figure 6: Anima at work.

4. *Search mechanism.* The search facility implements a pattern language that allows state information of interest to be searched on huge states and complex computation trees. The user only has to provide a filtering pattern (the query) that specifies the set of symbols that he/she wants to search for, and then all the states matching the query are automatically highlighted in the computation tree.

5. *Transition information.* Anima facilitates the inspection of a selected rewrite step $s \to t$ that occurs in the computation tree by underlining its redex in s and the reduced subterm in t. Some additional transition information is also displayed in the *transition information window* (e.g., the rule/equation applied, the rewrite position, and the computed substitution of the considered rewrite step) by right-clicking on the corresponding option.

7 Conclusions

The analysis of execution traces plays a fundamental role in many program analysis approaches, such as runtime verification, monitoring, testing, and specification mining. We have presented a parametrized exploration technique that can be applied to the inspection of rewriting logic computations and that can work in different ways. Three instances of the parameterized exploration scheme (an incremental stepper, an incremental partial stepper, and a forward trace slicer) have been formalized and implemented in the Anima tool, which is a novel program animator for RWL. The tool is useful for Maude programmers in two ways. First, it concretely demonstrates the semantics of the language, allowing the evaluation rules to be observed in action. Secondly, it can be used as a debugging tool, allowing the users to step forward and backward while slicing the trace in order to validate input data or locate programming mistakes.

References

[1] M. Alpuente, D. Ballis, M. Baggi, and M. Falaschi. A Fold/Unfold Transformation Framework for Rewrite Theories extended to CCT. In *Proc. 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2010*, pages 43–52. ACM, 2010.

[2] M. Alpuente, D. Ballis, J. Espert, and D. Romero. Model-checking Web Applications with Web-TLR. In *Proc. of 8th Int'l Symposium on Automated Technology for Verification and Analysis (ATVA 2010)*, volume 6252 of *LNCS*, pages 341–346. Springer-Verlag, 2010.

[3] M. Alpuente, D. Ballis, J. Espert, and D. Romero. Backward Trace Slicing for Rewriting Logic Theories. In *Proc. CADE 2011*, volume 6803 of *LNCS/LNAI*, pages 34–48. Springer-Verlag, 2011.

[4] M. Alpuente, D. Ballis, F. Frechina, and D. Romero. Backward Trace Slicing for Conditional Rewrite Theories. In *Proc. LPAR-18*, volume 7180 of *LNCS*, pages 62–76. Springer-Verlag, 2012.

[5] M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. Slicing-Based Trace Analysis of Rewriting Logic Specifications with iJulienne. In Matthias Felleisen and Philippa Gardner, editors, *Proc. of 22nd European Symposium on Programming, ESOP 2013*, volume 7792 of *Lecture Notes in Computer Science*, pages 121–124. Springer, 2013.

[6] M. Alpuente, D. Ballis, and D. Romero. Specification and Verification of Web Applications in Rewriting Logic. In *Formal Methods, Second World Congress FM 2009*, volume 5850 of *LNCS*, pages 790–805. Springer-Verlag, 2009.

[7] M. Baggi, D. Ballis, and M. Falaschi. Quantitative Pathway Logic for Computational Biology. In *Proc. of 7th Int'l Conference on Computational Methods in Systems Biology (CMSB '09)*, volume 5688 of *LNCS*, pages 68–82. Springer-Verlag, 2009.

[8] R. Bruni and J. Meseguer. Semantic Foundations for Generalized Rewrite Theories. *Theoretical Computer Science*, 360(1–3):386–414, 2006.

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude Manual (Version 2.6). Technical report, SRI Int'l Computer Science Laboratory, 2011. Available at: `http://maude.cs.uiuc.edu/maude2-manual/`.

[10] J. Clements, M. Flatt, and M. Felleisen. Modeling an Algebraic Stepper. In *Proc. 10th European Symposium on Programming*, volume 2028 of *LNCS*, pages 320–334. Springer-Verlag, 2001.

[11] F. Durán and J. Meseguer. A Maude Coherence Checker Tool for Conditional Order-Sorted Rewrite Theories. In *Proc. of 8th International Workshop on Rewriting Logic and Its Applications (WRLA'10)*, number 6381 in LNCS, pages 86–103. Springer-Verlag, 2010.

[12] J.W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume I, pages 1–112. Oxford University Press, 1992.

[13] N. Martí-Oliet and J. Meseguer. Rewriting Logic: Roadmap and Bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.

[14] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[15] J. Meseguer. The Temporal Logic of Rewriting: A Gentle Introduction. In *Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of his 65th Birthday*, volume 5065, pages 354–382, Berlin, Heidelberg, 2008. Springer-Verlag.

[16] A. Riesco, A. Verdejo, R. Caballero, and N. Martí-Oliet. Declarative Debugging of Rewriting Logic Specifications. In *Recent Trends in Algebraic Development Techniques, 19th Int'l Workshop, WADT 2008*, volume 5486 of *LNCS*, pages 308–325. Springer-Verlag, 2009.

[17] A. Riesco, A. Verdejo, and N. Martí-Oliet. Declarative Debugging of Missing Answers for Maude. In *21st Int'l Conference on Rewriting Techniques and Applications, RTA 2010*, volume 6 of *LIPIcs*, pages 277–294. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[18] TeReSe, editor. *Term Rewriting Systems*. Cambridge University Press, Cambridge, UK, 2003.

Automatic Inference of Term Equivalence in Term Rewriting Systems

Marco Comini[1] and Luca Torella[2]

[1] DIMI, University of Udine, Italy
marco.comini@uniud.it
[2] DIISM, University of Siena, Italy
luca.torella@unisi.it

Abstract

In this paper we propose a method to automatically infer algebraic property-oriented specifications from Term Rewriting Systems. Namely, having three semantics with suitable properties, given the source code of a TRS we infer a specification which consists of a set of *most general* equations relating terms that rewrite, for all possible instantiations, to the same set of constructor terms.

The semantic-based inference method that we propose can cope with non-constructor-based TRSs, and considers non-ground terms. Particular emphasis is put on avoiding the generation of "redundant" equations that can be a logical consequence of other ones.

1 Introduction

In the last years there has been a growing interest in the research on automatic inference of high-level specifications from an executable or the source code of a system. This is probably due to the fact that the size of software systems is growing over time and certainly one can greatly benefit from the use of automatic tools. There are several proposals, like [1, 7, 6], which have proven to be very helpful.

Specifications have been classified by their characteristics [8]. It is common to distinguish between *property-oriented* specifications and *model-oriented* or *functional* specifications. Property-oriented specifications are of higher description level than other kinds of specifications: they consist of an indirect definition of the system's behavior by stating a set of properties, usually in the form of axioms, that the system must satisfy [11, 10]. In other words, a specification does not represent the functionality of the program (the output of the system) but its properties in terms of relations among the operations that can be invoked in the program (i.e., identifies different calls that have the same behavior when executed). This kind of specifications is particularly well suited for program understanding: the user can realize non-evident information about the behavior of a given function by observing its relation to other functions. Moreover, the inferred properties can manifest potential symptoms of program errors which can be used as input for (formal) validation and verification purposes.

We can identify two mainstream approaches to perform the inference of specifications: glass-box and black-box. The glass-box approach [1] assumes that the source code of the program is available. In this context, the goal of inferring a specification is mainly applied to document the code, or to understand it. Therefore, the specification must be more succinct and comprehensible than the source code itself. The inferred specification can also be used to automate the testing process of the program or to verify that a given property holds [1]. The black-box approach [7, 6] works only by running the executable. This means that the only information used during the inference process is the input-output behavior of the program. In this setting, the inferred specification is often used to discover the functionality of the system (or services in a network) [6]. Although black-box approaches work without any restriction on the considered language – which is rarely the case in a glass-box approach – in general, they cannot *guarantee* the correctness of the results (whereas indeed semantics-based glass-box approaches can).

For this work, we developed a (glass-box) *semantic-based* algebraic specification synthesis method for the Term Rewriting Systems formalism that is *completely automatic*, i.e., needs only the source TRS to run. Moreover, *the outcomes are very intuitive* since specifications are sets of equations of the form $e_1 = e_2$, where e_1, e_2 are generic TRS expressions, so the user does not need any kind of extra knowledge to interpret the results. The underpinnings of our proposal are radically different from other works for functional programming, since the presence of *non-confluence* or *non-constructor-based* TRSs poses several additional problems.

1.1 Notations

We assume that the reader is familiar with the basic notions of term rewriting. For a thorough discussion of these topics, see [9]. In the paper we use the following notations. $\mathcal{V}$ denotes a (fixed) countably infinite set of variables and $\mathcal{T}(\Sigma, \mathcal{V})$ denotes the terms built over signature Σ and variables $\mathcal{V}$. $\mathcal{T}(\Sigma, \varnothing)$ are ground terms. Substitutions over $\mathcal{T}(\Sigma, \varnothing)$ are called ground substitutions. Σ is *partitioned* in $\mathcal{D}$, the *defined* symbols, and $\mathcal{C}$, the *constructor* symbols. $\mathcal{T}(\mathcal{C}, \mathcal{V})$ are *constructor terms*. Substitutions over $\mathcal{T}(\mathcal{C}, \mathcal{V})$ are said constructor substitutions. $C[t_1, \dots, t_n]$ denotes the replacement of $t_1, \dots, t_n$ in context C. A TRS $\mathcal{R}$ is a set of rules $l \to r$ where $l = f(t_1, \dots, t_n)$, $l, r \in \mathcal{T}(\Sigma, \mathcal{V})$, $var(r) \subseteq var(l)$ and $f \in \mathcal{D}$. $t_1, \dots, t_n$ are the argument patterns of $l \to r$ and need not necessarily be in $\mathcal{T}(\mathcal{C}, \mathcal{V})$, unlike in functional programming, where only *constructor-based* TRSs are considered (with $t_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$).

2 Many notions of equivalence

In the functional programming paradigm an equation $e_1 = e_2$ is typically *interpreted* as a property that holds for any *well-typed constructor ground* instance of the variables occurring in the equation. Namely, *for all bindings of variables with well-typed (constructor ground) terms* ϑ the constructor term computed for the calls $e_1\vartheta$ and $e_2\vartheta$ is the same. In functional programming we can consider only constructor instances because, by having constructor-based confluent TRSs, the set of values for non-constructor instances is the same as for constructor ones.

Differently from the functional programming case, the TRS formalism admits variables in initial terms and defined symbols in the patterns of the rules; moreover rules are evaluated non-deterministically and can rewrite to many constructor terms. Thus an equation can be interpreted in many different ways. We will discuss the key points of the problem by means of a (very simple) illustrative example.

Example 2.1 Consider the following (non constructor based) TRS R where we provide a pretty standard definition of the arithmetic operations `+`, `-` and (modulo) `%`:

```
0 + x -> x            x - 0 -> x              x % s(y) -> (x - s(y)) % s(y)
s(x) + y -> s(x+y)    s(x) - s(y) -> x - y    (0 - s(x)) % s(y) -> y - x
```

Note that, since the TRS formalism is untyped, a term like `0 + a` is admissible and is evaluated to constructor term `a`. ■

For this TRS, one *could* expect to have in its property-oriented specification equations like:

$$\texttt{(x + y) + z} = \texttt{x + (y + z)} \tag{2.1}$$
$$\texttt{x - (y + z)} = \texttt{(x - y) - z} \tag{2.2}$$
$$\texttt{(x - y) - z} = \texttt{(x - z) - y} \tag{2.3}$$
$$\texttt{0 + x} = \texttt{x} \tag{2.4}$$
$$\texttt{x + 0} = \texttt{x} \tag{2.5}$$
$$\texttt{x + y} = \texttt{y + x} \tag{2.6}$$

These equations, of the form $e_1 = e_2$, can be read as *the (observable) outcomes of* e_1 *are the same of* e_2. The first essential thing is to formalize the meaning of “observable outcomes” of the evaluation of an expression e (which contains variables). In the TRS formalism we have several possible combinations. First it is technically possible to *literally* interpret variables in e and decide to observe either the set of normal forms of e or the set of constructor terms of e. However, as can be quickly verified, only (2.4) is valid in either of these forms. Indeed, consider, for example, terms `x - (y + z)` and `(x - y) - z`. For both terms no rule can be applied, hence they are both (non constructor) normal forms and thus (2.2) is not literally valid. Clearly this is quite a radical choice. An interesting alternative would be to request that, for any possible interpretation of variables with any term, the expressions rewrite to the same set of constructor terms. Formally, by defining the rewriting behavior of a term t as

$$\mathcal{B}[\![t;\mathcal{R}]\!] \coloneqq \{\vartheta \cdot s \mid t\vartheta \xrightarrow[\mathcal{R}]{}^{*} s,\, s \in \mathcal{T}(\mathcal{C},\mathcal{V}),\, \vartheta\colon \mathcal{V} \to \mathcal{T}(\Sigma,\mathcal{V}),\, dom(\vartheta) \subseteq var(t) \tag{2.7}$$

we can interpret $e_1 = e_2$ as $e_1 =_{RB} e_2 :\Longleftrightarrow \mathcal{B}[\![e_1;\mathcal{R}]\!] = \mathcal{B}[\![e_2;\mathcal{R}]\!]$. In the following, we call this equivalence *rewriting behavior equivalence*. Actually, eqs. (2.1) to (2.4) are valid w.r.t. $=_{RB}$.

Note that if we would have chosen to use normal forms instead of constructor terms in (2.7) (i.e., $s \not\to$ instead of $s \in \mathcal{T}(\mathcal{C},\mathcal{V})$), then we would still have the same situation described before where only (2.4) holds.

The other equations (eqs. (2.5) and (2.6)) are not valid in this sense. For instance, $\mathcal{B}[\![\texttt{x + 0}]\!] = \{\{\texttt{x}/t\} \cdot \texttt{s}^i(\texttt{0}) \mid t \to^{*} \texttt{s}^i(\texttt{0})$ [1] while $\mathcal{B}[\![\texttt{x}]\!] = \{\{\texttt{x}/t\} \cdot v \mid t \xrightarrow[\mathcal{R}]{}^{*} v,\, v \in \mathcal{T}(\mathcal{C},\mathcal{V})$ (and then $\varepsilon \cdot \texttt{x} \in \mathcal{B}[\![\texttt{x}]\!] \smallsetminus \mathcal{B}[\![\texttt{x + 0}]\!]$). These equations are not valid essentially because we have variables which cannot be instantiated, but, if we consider ground constructor instancies which “trigger” in either term an evaluation to constructor terms, then we actually obtain the same constructor terms. For example for $t_1 = \texttt{x}$ and $t_2 = \texttt{x + 0}$, for all $\vartheta_i = \{\texttt{x}/\texttt{s}^i(\texttt{0})\}$, we have $\mathcal{B}[\![t_1\vartheta_i]\!] = \mathcal{B}[\![t_2\vartheta_i]\!] = \{\varepsilon \cdot \texttt{s}^i(\texttt{0})|$. Thus (2.5) holds in this sense.

Decidedly, also this notion of equivalence is interesting for the user. We can formalize it as $t_1 =_G t_2 :\Longleftrightarrow \forall\vartheta$ ground constructor. $\mathcal{B}[\![t_1\vartheta]\!] \cup \mathcal{B}[\![t_2\vartheta]\!] \subseteq \{\varepsilon \cdot t | t \in \mathcal{T}(\mathcal{C},\mathcal{V}) \Rightarrow \mathcal{B}[\![t_1\vartheta]\!] = \mathcal{B}[\![t_2\vartheta]\!]$ We will call it *ground constructor equivalence*. Note that $=_G$ is the only possible notion in the pure functional paradigm where we can just have evaluations of ground terms and where we have only confluent constructor-based TRSs. In this case the latter definition boils down to: two expressions are equivalent if all its ground constructor instances rewrite to the same constructor term. This fact allows one to have an intuition of the reason why the problem of specification synthesis is definitively more complex in the full TRS paradigm.

Since we do not consider only constructor-based TRSs, there is another very relevant difference w.r.t. the pure functional case. For instance, let us consider the TRS Q obtained by adding the rule `g((x - y) - z) -> x - (y + z)` to TRS R of Example 2.1. Let $t_1 = \texttt{x - (y + z)}$

[1] Here by $\texttt{s}^i(\texttt{0})$ we mean i repeated applications of `s` to `0`, including the degenerate case for $i = 0$.

and t_2 = `(x-y)-z`. We have $\mathcal{B}[\![t_1;Q]\!] = \mathcal{B}[\![t_2;Q]\!] = \{\{x/0, y/0, z/0\}\cdot 0, \{x/s(0), y/0, z/0\}\cdot s(0), \{x/s(0), y/s(0), z/0\}\cdot 0, \{x/s(0), y/0, z/s(0)\}\cdot 0, \dots |$. While the term `g(`$t_2$`)` has this same behavior, $\mathcal{B}[\![g(t_1);Q]\!] = \varnothing$.

In general, in the case of the TRS formalism, terms embedded within a context do not necessarily manifest the same behavior. Thus, it is also interesting to *additionally* ask to the equivalence notion $=_{RB}$ that the behaviors must be the same also when the two terms are embedded within any context. Namely, $e_1 =_C e_2 :\Longleftrightarrow \forall\,\text{context}\, C.\ \mathcal{B}[\![C[e_1];\mathcal{R}]\!] = \mathcal{B}[\![C[e_2];\mathcal{R}]\!]$ We call this equivalence *contextual equivalence*. We can see that $=_C$ is (obviously) stronger than $=_{RB}$, which is in turn stronger than $=_G$. Actually they are *strictly* stronger. Indeed, only eqs. (2.1) and (2.4) are valid w.r.t. $=_C$ (while eqs. (2.2) and (2.3) are not).

We believe that *all* these notions are interesting for the user, thus we formalize our notion of (algebraic) specification as follows.

Definition 2.2 *A specification $\mathcal{S}$ is a set of (sequences of) equations of the form $t_1 =_K t_2 =_K \dots =_K t_n$, with $K \in \{C, RB, G\}$ and $t_1, t_2, \dots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})$.*

Thus, for TRS Q we would get the following (partial) specification:

`(x + y) + z` $=_C$ `x + (y + z)` `0 + x` $=_C$ `x`

`x - (y + z)` $=_{RB}$ `(x - y) - z` `(x - y) - z` $=_{RB}$ `(x - z) - y`

`x + y` $=_G$ `y + x` `x + 0` $=_G$ `x`

`(x + y) % z` $=_G$ `((x % z) + (y % z)) % z` `(x + y) % y` $=_G$ `x % y`

`(x - y) % z` $=_G$ `(((x + z) % z) - (y % z)) % z`

In the following we present a first proposal of a semantics-based method that infers such specifications for TRSs and tackles the presented issues (and discuss about its limitations). It is an adaptation for the TRS paradigm of ideas from [2] for the Functional Logic paradigm. This adaptation is not straightforward, since the Functional Logic paradigm is quite similar to the Functional paradigm, but considerably different from the TRS paradigm. Moreover, this work significantly extends the inference process by tackling equations like $f(x,y) = f(y,x)$ which are really important.

3 Deriving specifications from TRSs

The methodology we are about to present is parametric w.r.t. three semantics evaluation functions which need to enjoy some properties. Namely,

$\mathcal{E}^{RB}[\![t;\mathcal{R}]\!]$ gives the *rewriting behavior* (RB) semantics of term t with (definitions from) TRS $\mathcal{R}$. This semantics has to be fully abstract w.r.t. rewriting behavior equivalence. Namely, we require that $\mathcal{E}^{RB}[\![t_1;\mathcal{R}]\!] = \mathcal{E}^{RB}[\![t_2;\mathcal{R}]\!] \Longleftrightarrow t_1 =_{RB} t_2$.

$\mathcal{E}^{C}[\![t;\mathcal{R}]\!]$ gives the *contextual* (C) semantics of the term t with the TRS $\mathcal{R}$. This semantics has to be fully abstract w.r.t. contextual equivalence. Namely, $\mathcal{E}^{C}[\![t_1;\mathcal{R}]\!] = \mathcal{E}^{C}[\![t_2;\mathcal{R}]\!] \Longleftrightarrow t_1 =_C t_2$.

$\mathcal{E}^{G}[\![t;\mathcal{R}]\!]$ gives the *ground* (G) semantics of the term t with the TRS $\mathcal{R}$. This semantics has to be fully abstract w.r.t. ground constructor equivalence. Namely, $\mathcal{E}^{G}[\![t_1;\mathcal{R}]\!] = \mathcal{E}^{G}[\![t_2;\mathcal{R}]\!] \Longleftrightarrow t_1 =_G t_2$.

The idea underlying the process of inferring specifications is that of computing the semantics of various terms and then identify all terms which have the same semantics. However, not all equivalences are as important as others, given the fact that many equivalences are simple logical consequences of others. For example, if $t_i =_C s_i$ then, for all constructor contexts C, $C[t_1, \dots, t_n] =_C C[s_1, \dots, s_n]$, thus the latter *derived* equivalences are uninteresting and should be omitted. Indeed, it would be desirable to synthesize the *minimal* set of equations from which, by deduction, all valid equalities can be derived. This is certainly a complex issue in testing approaches. With a semantics-based approach it is fairly natural to produce just the relevant equations. The (high level) idea is to proceed bottom-up, by starting from the evaluation of simpler terms and then newer terms are constructed (and evaluated) by using only semantically different arguments. Thus, by construction, only non-redundant equations are produced.

There is also another source of redundancy due to the inclusion of relations $=_K$. For example, since $=_C$ is the finer relation, $t =_C s$ implies $t =_{RB} s$ and $t =_G s$. To avoid the generation of coarser redundant equations, a simple solution is that of starting with $=_C$ equivalences and, once these are all settled, to proceed *only* with the evaluation of the $=_{RB}$ equivalences of *non* $=_C$ equivalent terms. Thereafter, we can evaluate the $=_G$ equivalences of *non* $=_{RB}$ equivalent terms.

Clearly, the *full* computation of a programs' semantics is not feasible in general. For the moment, for the sake of comprehension, we prefer to present the conceptual framework leaving out of the picture the issues related to decidability. We will show a possible decidable instance of the method in Section 3.1.

Let us describe in more detail the specification inference process. The input of the process consists of a TRS to be analyzed and two additional parameters: a *relevant* API, Σ^r, and a maximum term size, *max_size*. The *relevant* API allows the user to choose the operations in the program that will be present in the inferred specification, whereas the maximum term size limits the size of the terms in the specification. As a consequence, these two parameters tune the granularity of the specification, both making the process terminating and allowing the user to keep the specification concise and easy to understand.

The output consists of a set of equations represented by equivalence classes of terms (with distinct variables). Note that inferred equations may differ for the same program depending on the considered API and on the maximum term size. Similarly to other property-oriented approaches, the computed specification is complete up to terms of size *max_size*, i.e., it includes all the properties (relations) that hold between the operations in the relevant API and that are expressible by terms of size less or equal than *max_size*.

Terms are classified by their semantics into a data structure, which we call *classification*, consisting (conceptually) of a set of *equivalence classes* (*ec*) formed by

- $sem(ec)$: the semantics of (all) the terms in that class;
- $rep(ec)$: the *representative term* of the class ($rep(ec) \in terms(ec)$);
- $terms(ec)$: the set of terms belonging to that equivalence class;
- $epoch(ec)$: an integer to record the moment of introduction of that equivalence class.

The *representative term* is the term which is used in the construction of nested expressions when the equivalence class is considered. To output smaller equations it is better to choose the smallest term in the class (w.r.t. the function *size*), but any element of $terms(ec)$ can be used.

The inference process consists of successive phases, one for each kind of equality (in order of discriminating power, i.e., C, RB, G), as depicted in Figure 1.

Computation of the initial classification (epochs 0 and 1). The first phase of the algorithm, is the computation of the initial classification that is needed to compute the classification w.r.t. $=_C$. We initially create a classification which contains:

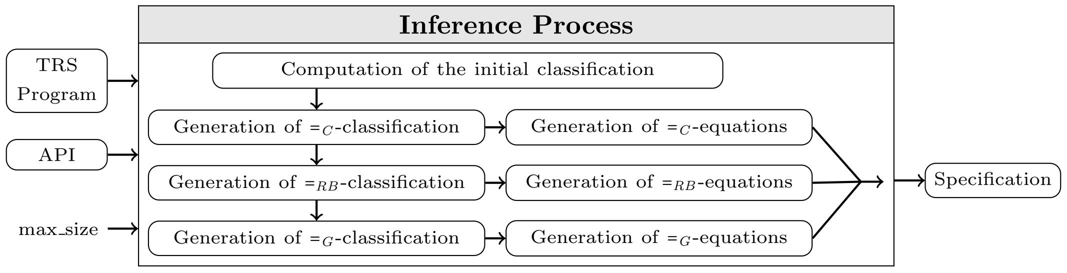

Inference Process
TRS Program
API
max_size
Computation of the initial classification
Generation of $=_C$-classification
Generation of $=_C$-equations
Generation of $=_{RB}$-classification
Generation of $=_{RB}$-equations
Generation of $=_G$-classification
Generation of $=_G$-equations
Specification

Figure 1: A general view of the inference process.

- one class for a free (logical) variable $\langle \mathcal{E}^C[\![x]\!], x, \{x\}, 0\rangle$;
- the classes for any constructor symbol, i.e., for all $c_{/n}$[2] $\in \mathcal{C}$ and $x_1, \dots, x_n$ distinct variables, $\langle \mathcal{E}^C[\![t]\!], t, \{t\}, 0\rangle$, where $t = c(x_1, \dots, x_n)$.

Then, for all symbols $f_{/n}$ of the relevant API, Σ^r, and distinct variables $x_1, \dots, x_n$, we *add to classification* the term $t = f(x_1, \dots, x_n)$ with semantics $s = \mathcal{E}^C[\![t; \mathcal{R}]\!]$ and epoch $m = 1$. This operation, denoted $addEC(t, s, m)$ is the most delicate part of the method. If we would not consider the generation of "permuted" equations like $f(x, y) = f(y, x)$, then the whole activity would just boil down to look for the presence of s in some equivalence class and then updating the classification accordingly (like in [2]). Handling of permutations cannot be done by naïvely adding all (different) permutations of s. In this way we would generate many redundant equations. Consider for instance the TRS Q

```
f(a,b,c) -> 0          f(a,c,b) -> 0          f(c,a,b) -> 0
f(b,a,c) -> 0          f(b,c,a) -> 0          f(c,b,a) -> 0
```

From the minimal set $f(x, y, z) =_C f(y, x, z) =_C f(x, z, y)$ we can generate all other valid permutations, like $f(x, y, z) =_C f(y, z, x) =_C f(z, x, y)$. Thus *in this case* we should only generate permutated equations where we just swap two variables. However, for the TRS R

```
f(a,b,c) -> 0          f(b,c,a) -> 0          f(c,a,b) -> 0
```

$f(x, y, z) \neq_C f(y, x, z) \neq_C f(x, z, y)$ but $f(x, y, z) =_C f(y, z, x)$. Moreover $f(x, y, z) =_C f(z, x, y)$ can be deduced by this. Thus *in this case* we should only generate permutated equations with a rotation of three variables, since all other rotations of three variables are just a consequence.

Thus, not all permutations have to be considered while adding a semantics to a classification, and it is not necessary to look for all permutations within the semantics already present in the classification. To generate only a minimal set of necessary permutated equations we need to consider, for each k variables, a set Π^n_k of *generators* of the permutations of n variables which do not move $n - k$ variables (note that $\Pi^n_1 = \{id\}$). Then, for a term $t = f(x_1, \dots, x_n)$, we start, sequentially for i from 1 to n, and look if, for some $\pi \in \Pi^n_i$, we have an equivalence class ec in the current classification whose semantics coincides with $s\pi$ (i.e., $ec = \langle s\pi, t', T, m'\rangle$). If it is found, then the term $t\pi^{-1}$ is added to the set of terms in ec (i.e., ec is transformed in $ec' = \langle s\pi, t', T \cup \{t\pi^{-1}\}, m'\rangle$) and we stop. Otherwise, we iterate with next i. If all fails, a new equivalence class $\langle s, t, T, m\rangle$ has to be created, but we have to determine the right term set

[2]Following the standard notation $f_{/n}$ denotes a function f of arity n.

T (to possibly generate equations like $f(x,y) = f(y,x)$). We initially start with $T = \{t\}$ and $j = n$, and sequentially for i from 2 to j, we check if, for some $\pi \in \Pi_i^j$, we have $s = s\pi$. If we find it, we add $t\pi$ to T, and decrement j by $i-1$ (i variables are now used in a valid equation and thus from this moment on $i-1$ variables have no longer to be considered). Otherwise we continue with next i. The final T is used for the new equivalence class.

For example, for $t = f(x,y,z)$ in TRS Q, we start with variables $\{x,y,z\}$ and $T = \{t\}$. Now consider, for instance, $\pi = (x\,y) \in \Pi_2^3$. Since $\mathcal{E}^C[\![t]\!] = \mathcal{E}^C[\![t\pi]\!]$ then $T = \{f(x,y,z), f(y,x,z)\}$ and then we drop x and remain with variables $\{y,z\}$. Now we have only $\pi' = (y\,z) \in \Pi_2^2$, and since $\mathcal{E}^C[\![t]\!] = \mathcal{E}^C[\![t\pi']\!]$ then $T = \{f(x,y,z), f(y,x,z), f(x,z,y)\}$, we drop y and (thus) finish. Instead, for $t = f(x,y,z)$ in TRS R we start with variables $\{x,y,z\}$ and $T = \{t\}$. Now consider, for instance, $\pi_1 = (x\,y) \in \Pi_2^3$. Since $\mathcal{E}^C[\![t]\!] \neq \mathcal{E}^C[\![t\pi_1]\!]$ then we consider $\pi_2 = (y\,z) \in \Pi_2^3$, and since $\mathcal{E}^C[\![t]\!] \neq \mathcal{E}^C[\![t\pi_2]\!]$ we increase i. Now we have only $\pi_3 = (x\,y\,z) \in \Pi_3^3$, and since $\mathcal{E}^C[\![t]\!] = \mathcal{E}^C[\![t\pi_3]\!]$ then $T = \{f(x,y,z), f(y,z,x)\}$, we drop x, y and (thus) finish.

Generation of $=_C$ classification (epochs 2 and above). The second phase of the algorithm, is the (iterative) computation of the successive epochs, until we complete the construction of the classification of terms w.r.t. $=_C$. At each iteration (with epoch k), for all symbols $f_{/n}$ of the relevant API Σ^r, we select from the current classification all possible combinations of n equivalence classes $ec_1, \dots, ec_n$ such that at least one ec_i was newly produced in the previous iteration (i.e., whose epoch is $k-1$). We build the term $t = f(rep(ec_1), \dots, rep(ec_n))$ which, by construction, has surely not been considered yet. Then, if $size(t) \leq max\_size$, we compute the semantics $s = \mathcal{E}^C[\![t; \mathcal{R}]\!]$ and update the current classification by *adding to classification* the term t and its semantics s ($addEC(t,s,k)$) as described before.

If we have produced new equivalence classes then we continue to iterate. This phase eventually terminates because at each iteration we consider, by construction, terms which are different from those already existing in the classification and whose size is strictly greater than the size of its subterms (but the size is bounded by max_size).

The following example illustrates how the iterative process works:

Example 3.1 Let us use the program R of Example 2.1 and choose as relevant API the functions `+`, `-` and `%`. In the first step, the terms

$$t_{1.1} = \texttt{x + y} \qquad t_{1.2} = \texttt{x - y} \qquad t_{1.3} = \texttt{x \% y}$$

are built. Since (all permutations of) the semantics of all these terms are different, and different from the other semantics already in the initial classification, three new classes are added to the initial classification.

During the second iteration, the following two terms (among others) are built:

- the term $t_{2.1}$ = `(x' + y') + y` is built as the instantiation of `x` in $t_{1.1}$ with (a renamed apart variant of) $t_{1.1}$, and
- the term $t_{2.2}$ = `x + (x' + y')` as the instantiation of `y` in $t_{1.1}$ with $t_{1.1}$.

The semantics of these two terms is the same s, but it is different from the semantics of the existing equivalence classes. Thus, during this iteration (at least) the new equivalence class $ec' := \langle s, t_{2.1}, \{t_{2.1}, t_{2.2}\}, n\rangle$ is computed. Hereafter, only the representative of the class will be used for constructing new terms. Since we have chosen $t_{2.1}$ instead of $t_{2.2}$ as the representative, terms like `(x + (x' + y')) % z` will never be built. ■

Thanks to the closedness w.r.t. context of the semantics, this strategy for generating terms is *safe*. In other words, when we avoid to build a term, it is because it is not able to produce a behavior different from the behaviors already included by the existing terms, thus we are not losing completeness.

Generation of the $=_C$ specification Since, by construction, we have avoided much redundancy thanks to the strategy used to generate the equivalence classes, we now have only to take each equivalence class with more than one term and generate equations for these terms.

Generation of $=_{RB}$ equations The third phase of the algorithm works on the former classification by first transforming each equivalence class ec by replacing the C-semantics $sem(ec)$ with $\mathcal{E}^{RB}[\![rep(ec); \mathcal{R}]\!]$ and $terms(ec)$ with the (singleton) set $\{rep(ec)\}$. After the transformation, some of the previous equivalence classes which had different semantic constructor terms may now have the same RB-semantics and then we merge them, making the union of the term sets $terms(ec)$.

Thanks to the fact that, before merging, all equivalence classes were made of just singleton term sets, we cannot generate (again) equations $t_1 =_{RB} t_2$ when an equation $t_1 =_C t_2$ had been already issued. Let us clarify this phase by an example.

Example 3.2 Assume we have a classification consisting of three equivalence classes with C-semantics s_1, s_2 and s_3 and representative terms t_{11}, t_{22} and t_{31}:

$$ec_1 = \langle s_1, t_{11}, \{t_{11}, t_{12}, t_{13}\}\rangle \qquad ec_2 = \langle s_2, t_{22}, \{t_{21}, t_{22}\}\rangle \qquad ec_3 = \langle s_3, t_{31}, \{t_{31}\}\rangle$$

We generate equations $t_{11} =_C t_{12} =_C t_{13}$ and $t_{21} =_C t_{22}$.

Now, assume that $\mathcal{E}^{RB}[\![t_{11}]\!] = w_0$ and $\mathcal{E}^{RB}[\![t_{22}]\!] = \mathcal{E}^{RB}[\![t_{31}]\!] = w_1$. Then (since t_{12}, t_{13} and t_{21} are removed) we obtain the new classification

$$ec_4 = \langle w_0,\, t_{11},\, \{t_{11}\},\, n\rangle \qquad ec_5 = \langle w_1,\, t_{22},\, \{t_{22}, t_{31}\},\, n\rangle$$

Hence, the only new equation is $t_{22} =_{RB} t_{31}$. Indeed, equation $t_{11} =_{RB} t_{12}$ is uninteresting, since we already know $t_{11} =_C t_{12}$ and equation $t_{21} =_{RB} t_{31}$ is redundant (because $t_{21} =_C t_{22}$ and $t_{22} =_{RB} t_{31}$). ■

The resulting (coarser) classification is then used to produce the $=_{RB}$ equations, as done before, by generating equations for all non-singletons term sets.

Generation of the $=_G$ equations In the last phase, we transform again the classification by replacing the RB-semantics with the G-semantics (and $terms(ec)$ with the set $\{rep(ec)\}$). Then we merge eventual equivalence classes with the same semantics and, finally, we generate $=_G$ equations for non singleton term sets.

Theorem 3.3 (Correctness) *For all equations $e_1 = e_2$ generated in the second, third and forth phase we have that $e_1 =_C e_2$, $e_1 =_{RB} e_2$ and $e_1 =_G e_2$, respectively.*

Proof. By construction of equivalence classes, in the second phase an equation $t_1 = t_2$ is generated if and only if the semantics $\mathcal{E}^C[\![t_1; \mathcal{R}]\!] = \mathcal{E}^C[\![t_2; \mathcal{R}]\!]$. Then, since $\mathcal{E}^C[\![t_1; \mathcal{R}]\!] = \mathcal{E}^C[\![t_2; \mathcal{R}]\!] \iff t_1 =_C t_2$, the first part of thesis follows immediately. Then, by the successive trasformation and reclassification, in the third phase we issue an equation $t_1 = t_2$ if and only if $\mathcal{E}^{RB}[\![t_1; \mathcal{R}]\!] = \mathcal{E}^{RB}[\![t_2; \mathcal{R}]\!]$. Then, since $\mathcal{E}^{RB}[\![t_1; \mathcal{R}]\!] = \mathcal{E}^{RB}[\![t_2; \mathcal{R}]\!] \iff t_1 =_{RB} t_2$, we have the second part of thesis. The proof of the third part of thesis, since $\mathcal{E}^G[\![t_1; \mathcal{R}]\!] = \mathcal{E}^G[\![t_2; \mathcal{R}]\!] \iff t_1 =_G t_2$, is analogous. □

Example 3.4 Let us recall (again) the program R of Example 2.1. The terms `x + y` and `y + x`, before the transformation, belong to two different equivalence classes (since their rewriting behavior is different). Anyway, after the transformation, the two classes are merged since their G-semantics is the same, namely $\big\{\{x/0, y/0\} \cdot 0, \{x/s(0), y/0\} \cdot s(0), \{x/0, y/s(0)\} \cdot s(0), \ldots\big|$. ■

We now present some examples to show how our proposal deals with non-contructor based or non-confluent TRS, plus some to show that we can infer several non-trivial equations.

Example 3.5 Consider the definition for the boolean data type with constructor terms `true` and `false` and boolean operations `and`, `or`, `not` and `imp`:

```
and(true,x) -> x                    not(true) -> false
and(false,x) -> false               not(false) -> true
or(true,x) -> true                  imp(false,x) -> true
or(false,x) -> x                    imp(true,x) -> x
```

This is a pretty standard "short-cut" definition of boolean connectives. With our method we get the following equations:

$$\texttt{not(or(x,y))} =_C \texttt{and(not(x),not(y))} \qquad \texttt{imp(x,y)} =_C \texttt{or(not(x),y)}$$
$$\texttt{not(and(x,y))} =_C \texttt{or(not(x),not(y))} \qquad \texttt{not(not(x))} =_G \texttt{x}$$
$$\texttt{and(x,and(y,z))} =_C \texttt{and(and(x,y),z)} \qquad \texttt{and(x,y)} =_G \texttt{and(y,x)}$$

■

Example 3.6 Let us consider the following *non-constructor based* TRS implementing some operations over the naturals in Peano notation.

```
x - 0 -> x                          chk(0) -> 0
s(x) - s(y) -> x - y                chk(s(x)) -> s(x)
g(x) -> chk(x - s(x))               chk(0 - s(x)) -> err
```

The definition of `-` is a standard definition of the minus operation over naturals. The `chk` function simply returns the natural passed as argument, or returns `err` if the argument is a not defined subtraction. It is easy to see that the TRS is not constructor-based because of the presence of the `-` in the pattern of a rule. The artificial function `g`, which checks if the subtraction of a number by its successor is a natural number, is doomed to return `err`. With our classification we actually derive equation $\texttt{g(x)} =_C \texttt{err}$. ■

Example 3.7 Let us consider the following (artificial) *non-orthogonal* TRS.

```
coin -> 0               d(x) -> g(x,x)          t(x) -> k(x,x,x)
coin -> 1               g(0,1) -> true          k(1,0,1) -> true
```

The `coin` function can return both `0` and `1`. The functions `d` and `t` call an auxiliary function, duplicating and triplicating (respectively) the variable received as argument. Functions `f` and `g` require a specific combination of `0` and `1` to return the constructor term `true`. Notice that, to reach the constructor term `true` from `d` and `t` respectively, it is necessary to use a non deterministic function able to rewrite to both `0` and `1`. Some of the inferred equations for this TRS are $\texttt{t(x)} =_C \texttt{d(x)}$ and $\texttt{t(coin)} =_C \texttt{g(coin,coin)} =_C \texttt{k(coin,coin,coin)} =_C \texttt{d(coin)}$.
■

Example 3.8 Let us consider the following TRS that computes the double of numbers in Peano notation:

```
double(0) -> 0                         plus(0,y) -> y
double(s(x)) -> s(s(double(x)))        plus(s(x),y) -> s(plus(x,y))
dbl(x) -> plus(x,x)
```

Some of the inferred equations for the TRS are:

$$\texttt{dbl(dbl(double(x)))} =_G \texttt{dbl(double(dbl(x)))} \tag{3.1}$$

$$\texttt{double(x)} =_G \texttt{dbl(x)} \tag{3.2}$$

$$\texttt{dbl(dbl(x))} =_G \texttt{dbl(double(x))} =_G \texttt{double(dbl(x))} =_G \texttt{double(double(x))} \tag{3.3}$$

$$\texttt{plus(double(x),y)} =_G \texttt{plus(dbl(x),y)} \tag{3.4}$$

We can observe that all equations hold with the $=_G$ relation, and not with the $=_{RB}$ relation. This is due to the fact that the two functions `dbl` and `dbl`, even if at a first sight could seem to evaluate the same constructor terms, can behave differently. For instance, if we add to the TRS the two rules `coin -> 0` and `coin -> s(0)` we can notice that the terms `double(coin)` and `dbl(coin)` return different constructor terms. This is due to the non determinism of the `coin` function that exploits the right non-linearity of function `dbl`. While `double(coin)` evaluates to `0` and `s(s(0))`, the term `dbl(coin)` can be reduced even to `s(0)` (by `dbl(coin)` $\to$ `plus(coin,coin)` $\to^2$ `plus(0,s(0))` $\to$ `s(0)`).

This characteristic of the program is not easy to realize by just looking at the code. ■

Example 3.9 Let us consider the following TRS defining two functions over an extension of the Peano notation able to handle negative integers:

```
abs(-(x)) -> abs(x)                    f(-(-(x))) -> f(x)
abs(s(x)) -> s(x)                      f(0) -> 0
abs(0) -> 0                            f(s(x)) -> s(x)
                                       f(-(s(x))) -> 0
```

Function `abs` is a standard definition of the absolute value; function `f` returns its input if it is a positive number, and `0` if it is not. Some of the inferred equations are `f(f(x))` $=_C$ `abs(f(x))` and `f(abs(x))` $=_C$ `abs(abs(x))`. ■

Example 3.10 Let us consider the following program which implements a two-sided queue in a (non-trivial) efficient way. The queue is implemented as two lists where the first list corresponds to the first part of the queue and the second list is the second part of the queue reversed. The `inl` function adds the new element to the head of the first list, whereas the `inr` function adds the new element to the head of the second list (the last element of the queue). The `outl` (`outr`) function drops one element from the left (right) list, unless the list is empty, in which case it reverses the other list and then swaps the two lists before removal.

```
new -> Q(Nil,Nil)                      outr(Q(xs,Nil)) ->
inl(x,Q(xs,ys)) -> Q(:(x,xs),ys)               Q(Nil,tail(rev(xs)))
inr(x,Q(xs,ys)) -> Q(xs,:(x,ys))       outr(Q(xs,:(y,ys))) -> Q(xs,ys)
outl(Q(Nil,ys)) ->                     null(Q(:(x,xs),ys)) -> False
        Q(tail(rev(ys)),Nil)           null(Q(Nil,:(x,xs))) -> False
outl(Q(:(x,xs),ys)) -> Q(xs,ys)        null(Q(Nil,Nil)) -> True
```

```
tail(:(x,xs)) -> xs                    rv'(Nil,ys) -> ys
rev(xs) -> rv'(xs,Nil)                 rv'(:(x,xs),ys) -> rv'(xs,:(x,ys))
```

With our method (amongst others) we derive:

$$\texttt{null(new)} =_C \texttt{True} \tag{3.5}$$

$$\texttt{new} =_C \texttt{outl(inl(x,new))} =_C \texttt{outr(inr(x,new))} \tag{3.6}$$

$$\texttt{outl(inl(x,q))} =_C \texttt{outr(inr(x,q))} \tag{3.7}$$

$$\texttt{inr(x,inl(y,q))} =_C \texttt{inl(y,inr(x,q))} \tag{3.8}$$

$$\texttt{inl(x,outl(inl(y,q)))} =_C \texttt{outr(inl(x,inr(y,q)))} \tag{3.9}$$

$$\texttt{null(inl(x,new))} =_C \texttt{null(inr(x,new))} =_C \texttt{False} \tag{3.10}$$

We can see different kinds of non-trivial equations: eqs. (3.6) and (3.7) state that adding and removing one element produces always the same result independently from the side in which we add and remove it. Equations (3.8) and (3.9) show a sort of *restricted* commutativity between functions. ∎

3.1 An effective instance of the presented method

In a semantics-based approach, one of the main problems to be tackled is effectiveness. The semantics of a program is in general infinite and thus some approximation has to be used in order to have a terminating method. To experiment on the validity of our proposal we started by using a novel (condensed) fixpoint semantics which we have developed for left-linear TRSs that is fully abstract w.r.t. $=_C$ [3]. Such semantics is defined as the fixpoint of an immediate consequences operator $\mathcal{P}[\![\mathcal{R}]\!]$. We have opted for this semantics because it has some properties which are very important from a pragmatical point of view:

- it is condensed, meaning that denotations are the smallest possible (between all those semantics which induce the same program equivalence). This is a very relevant (if not essential) feature to develop a semantic-based tool which has to *compute* the semantics.
- The semantics $\mathcal{E}^{RB}$ can be obtained directly by transforming the $\mathcal{E}^{C}$ semantics, concretely just by loosing internal structure. Therefore, no (costly) computation of $\mathcal{E}^{RB}$ is needed.

We have implemented the basic functionality of the proposed methodology in a prototype written in Haskell, TRSynth, available at http://safe-tools.dsic.upv.es/trsynth (for a detailed description see [4]). The implementation of $=_G$ equality is still ongoing work, because we are lacking of a suitable implementation of the G-semantics.

To achieve termination, the prototype computes a fixed number k of steps of $\mathcal{P}[\![\mathcal{R}]\!]$. Then, it proceeds with the classification as described in Section 3. Clearly, in presence of terms with infinite solutions, with such a rough approximation we may loose both correctness (by mistakenly equating terms which become semantically different after k iterates) and completeness (by mistakenly not equating terms which will become semantically equivalent). Nevertheless the results are encouraging. For instance TRSynth detects all $=_C$ and $=_{RB}$ which we showed in examples (except of Example 3.8 because of a bug in the computation of the semantics).

[3]The writing of articles related to the formal definition of this semantics is in progress [3].

4 Conclusions and future work

This paper discusses about the issues that arise for the automatic inference of high-level, property-oriented (algebraic) specifications because of non-confluent or non-constructor based TRS. Then, a first proposal which overcomes these issues is presented.

Our method computes a concise specification of program properties from the source code of a TRS. We hope to have convinced the reader (with all examples) that we reached our main goal, that is, to get a concise and clear specification (that is useful for the programmer in order to discover unseen properties or detect possible errors).

We have developed a prototype that implements the basic functionality of the approach. We are aware that many other attempts to guarantee termination could be used in other instancies of the presented method. Certainly, given our know-how, in the future we will experiment with abstractions obtained by abstract interpretation [5] (our novel semantics itself has been obtained as an abstract interpretation). Actually we already have an ongoing work to implement the $depth(k)$ abstract version of our semantics. In the $depth(k)$ abstraction, terms (occurring in the nodes of the semantic trees) are "cut" at depth k by replacing them with *cut variables*, distinct from program variables. Hence, for a given signature Σ, the universe of abstract semantic trees is finite (although it increases exponentially w.r.t. k). Therefore, the finite convergence of the computation of the abstract semantics is guaranteed.

References

[1] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'02)*, pages 4–16, New York, NY, USA, 2002. Acm.

[2] G. Bacci, M. Comini, M. A. Feliú, and A. Villanueva. Automatic Synthesis of Specifications for First Order Curry Programs. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*, pages 25–34, New York, NY, USA, 2012. ACM.

[3] M. Comini and L. Torella. A Condensed Goal-Independent Fixpoint Semantics Modeling the Small-Step Behavior of Rewriting. Technical Report DIMI-UD/01/2013/RR, Dipartimento di Matematica e Informatica, Università di Udine, 2013.

[4] M. Comini and L. Torella. TRSynth: a Tool for Automatic Inference of Term Equivalence in Left-linear Term Rewriting Systems. In E. Albert and S.-C. Mu, editors, *PEPM '13, Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 67–70. Acm, 2013.

[5] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, Los Angeles, California, January 17–19*, pages 238–252, New York, NY, USA, 1977. ACM Press.

[6] C. Ghezzi, A. Mocci, and M. Monga. Synthesizing intensional behavior models by graph transformation. In *31st International Conference on Software Engineering (ICSE'09)*, pages 430–440, 2009.

[7] J. Henkel, C. Reichenbach, and A. Diwan. Discovering Documentation for Java Container Classes. *IEEE Transactions on Software Engineering*, 33(8):526–542, 2007.

[8] A. A. Khwaja and J. E. Urban. A property based specification formalism classification. *The Journal of Systems and Software*, 83:2344–2362, 2010.

[9] TeReSe, editor. *Term Rewriting Systems*. Cambridge University Press, Cambridge, UK, 2003.

[10] H. van Vliet. *Software Engineering–Principles and Practice*. John Wiley, 1993.

[11] J. M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):10–24, 1990.

A Condensed Goal-Independent Fixpoint Semantics Modeling the Small-Step Behavior of Rewriting

Marco Comini[1] and Luca Torella[2]

[1] DIMI, University of Udine, Italy — marco.comini@uniud.it
[2] DIISM, University of Siena, Italy — luca.torella@unisi.it

Abstract

In this paper we present a novel condensed narrowing-like semantics that contains the minimal information which is needed to describe compositionally all possible rewritings of a term rewriting system. We provide its goal-dependent top-down definition and, more importantly, an equivalent goal-independent bottom-up fixpoint characterization.

We prove soundness and completeness w.r.t. the small-step behavior of rewriting for the *full class* of term rewriting systems.

1 Introduction

Nowadays the formalism of Term Rewriting Systems (TRSs) is used, besides for functional programming, also for many other applications (like specification of communication protocols, to mention one). There has been a lot of research on the development of tools for the formal verification and (in general) automatic treatment/manipulation of TRSs. Within the proposals there are semantics-based approaches which can guarantee correctness by construction. However they cannot employ directly the construction of the semantics, since in general it is infinite. Thus some kind of approximation has to be used.

Given the potentiality of application of the TRS formalism, we have turned our attention toward the development of semantics-based TRS manipulation tools with the intention to use Abstract Interpretation theory as fundament to devise semantics approximations correct by construction. However, as also noted by [9], defining a suitable (concrete) semantics is usually the first crucial step in adapting the general methodology of Abstract Interpretation to the semantic framework of the programming language at hand. When a concrete semantics is used to define, via abstract interpretation, abstract (approximate) semantics to be employed to develop semantics-based manipulation tools, it is *particularly* relevant if it is *condensed* and defined compositionally. In the literature, a semantics is said to be *condensing* when the semantics of an instance of an expression (term, goal, call) can be obtained with a semantic operation directly from the semantics of the (un-instantiated) expression. In such a situation, only the semantics for most general expressions can be maintained in denotations. We say that a semantics is *condensed* when the denotations themselves do not contain redundancy, i.e., when it is not possible to semantically derive the components of a denotation from the other components. Indeed, the abstract semantics operations which are obtained from a condensed concrete semantics involve the use of the join operation (of the abstract domain) at each iteration in parallel onto all components of rules, instead of using several subsequent applications for all components. This has a twofold benefit. On one side, it speeds up convergence of the abstract fixpoint computation. On the other side, it considerably improves precision.

In [2], we developed an automatic debugging methodology for the TRS formalism based on abstract interpretation of the big-step rewriting semantics that is most commonly considered in functional programming, i.e., the set of constructor terms/normal forms. However, the resulting tool was inefficient. The main reason for this inefficiency is because the chosen concrete semantics is not condensing and thus, because of its accidental high redundancy, it causes the algorithms to use and produce much redundant information at each stage. In contrast, the same methodology gave good results in [5] because it was applied to a condensed semantics.

The constructor terms/normal forms semantics is not condensed because it contains all possible rewritings, but there are many possible rewritings which can be obtained by some other ones, since rewriting is closed under substitution (stability) and replacement ($t \xrightarrow[\mathcal{R}]{}^{*} s$ implies $C[t]_p \xrightarrow[\mathcal{R}]{}^{*} C[s]_p$). Thus, in [1] we tried to directly devise a semantics, fully abstract w.r.t. the big-step rewriting semantics, with the specific objective to avoid all redundancy while still characterizing the rewritings of any term. In particular we searched a semantics which

- has a compositional goal-independent definition,
- is the fixpoint of a bottom-up construction,
- is as condensed as possible.

Unfortunately (in [1]) we just partially achieved this goal since the semantics is defined only for some classes of TRSs. In the meantime, for a (quite) different language, in [4, 3] we obtained—for the full language—a semantics with the mentioned characteristics, by following a different approach:

1. Define a denotation, *fully abstract* w.r.t. the *small-step* behavior of evaluation of expressions, which enjoys the mentioned properties.
2. Obtain by abstraction of this small-step semantics a denotation (which enjoys the mentioned properties) correct w.r.t. the *big-step* behavior.

This approach has the additional advantage that the small-step semantics can be reused also to develop other semantics more concrete than the *big-step* one (for instance semantics which can model functional dependencies that are suitable to develop pre-post verification methods).

Unfortunately in the case of the TRS formalism we do not have a suitable *small-step* semantics to start with. For Curry we defined the small-step semantics by collecting just the most general traces of the small-step operational semantics, which correspond (in our case) to the rewriting derivations of the terms $f(x_1, \dots, x_n)$. The problem is that we cannot obtain, just from the traces of all $f(x_1, \dots, x_n)$, the rewriting derivations of all (nested) terms, without using again (directly or indirectly) the rewriting mechanism. In fact, usually $f(x_1, \dots, x_n)$ is immediately a normal form, because we cannot instantiate variables; however, there are many instances which can trigger the rules. *Narrowing* [7] can seem a possible solution to this problem but we have an issue related to the interference of non-confluence (i.e., non-determinism) with non-linearity, as shown by this example.

Example 1.1

Let us consider the following TRS $\mathcal{R}$:

$$\begin{array}{lll} \mathit{coin} \to \mathit{Tail} & \mathit{Head} \neq \mathit{Tail} \to \mathit{True} & \mathit{diff}(x) \to x \neq x \\ \mathit{coin} \to \mathit{Head} & \mathit{Tail} \neq \mathit{Head} \to \mathit{True} & \end{array}$$

We have rewriting derivations $\mathit{diff}(x) \to x \neq x \not\to$, $\mathit{diff}(\mathit{Head}) \to \mathit{Head} \neq \mathit{Head} \not\to$, $\mathit{diff}(\mathit{Tail}) \to \mathit{Tail} \neq \mathit{Tail} \not\to$, while $\mathit{diff}(\mathit{coin}) \to^{*} \mathit{True}$. Moreover, we have the narrowing derivation $\mathit{diff}(x) \overset{\varepsilon}{\rightsquigarrow} x \neq x \not\rightsquigarrow$.

Narrowing can instantiate variables (according to rules), but a variable is instantiated with the same term in all of its occurrences (it would make little sense to do differently, for a top-down resolution mechanism). However, Example 1.1 shows that it is not possible to retrieve that $\mathit{diff}(\mathit{coin}) \to^* \mathit{True}$ from all possible narrowing derivations of $\mathit{diff}(x)$, since the only narrowing derivation (of $\mathit{diff}(x)$) does not reach True.

In this paper we define a variation of narrowing (linearizing narrowing) which admits different instances of variables with multiple occurrences. With linearizing narrowing we define a denotation, *fully abstract* w.r.t. the *small-step* behavior of rewriting, which enjoys the mentioned properties *for generic TRSs without restrictions*. The outline to achieve this is the following.

- We gather all linearizing narrowing derivations into trees (Definition 3.10).
- We show that all possible rewritings can be reconstructed from linearizing narrowing trees (Theorem 3.14).
- We define top-down condensed denotations $\mathcal{O}[\![\mathcal{R}]\!]$ by collecting just the linearizing narrowing trees of most general terms ($f(\overrightarrow{x_n})$) and we prove that, with a suitable semantic evaluation function $\mathcal{E}$, we can reconstruct any linearizing narrowing tree starting from $\mathcal{O}[\![\mathcal{R}]\!]$ (Theorem 3.23).
- By using $\mathcal{E}$ we define a (bottom-up) immediate consequence operator whose least fixpoint $\mathcal{F}[\![\mathcal{R}]\!]$ is equal to $\mathcal{O}[\![\mathcal{R}]\!]$ (Theorem 3.30). Thus from $\mathcal{F}[\![\mathcal{R}]\!]$ we can reconstruct all possible rewritings of $\mathcal{R}$ and we have full abstraction w.r.t. the rewriting behavior (Corollary 3.31).

Note that the proofs of all results are in the appendix.

2 Preliminaries

We assume that the reader is familiar with the basic notions of term rewriting. For a thorough discussion of these topics, see [10]. In the paper we use the following notions and notations.

We write $\overrightarrow{o_n}$ for the *list* of syntactic objects $o_1, \dots, o_n$. Given a monotonic function $F \colon \mathcal{L} \to \mathcal{L}$, over lattice $\mathcal{L}$ whose bottom is $\bot$ and *lub* is $\bigsqcup$, by $F{\uparrow}k$ we denote function $\lambda x. F^k(x)$ and by $F{\uparrow}\omega$ function $\lambda x. \bigsqcup\{F^k(x) | k \in \mathbb{N}$. By $\mathit{lfp}(F)$ we denote the least fixed point of F (and recall that, for a continuos F, $\mathit{lfp}(F) = F{\uparrow}\omega$).

Terms and Substitutions

Σ denotes a signature and $\mathcal{V}$ denotes a (fixed) countably infinite set of variables. $\mathcal{T}(\Sigma, \mathcal{V})$ denotes the terms built over signature Σ and variables $\mathcal{V}$. Σ is *partitioned* in $\mathcal{D}$, the *defined* symbols (also called operations), and $\mathcal{C}$, the *constructor* symbols (also called data constructors). $\mathcal{T}(\mathcal{C}, \mathcal{V})$ are called *constructor terms*. The set of variables occurring in a term t is denoted by $\mathit{var}(t)$, while the sequence (in order) of variables occurring in a term t is denoted by $\overrightarrow{\mathit{var}}(t)$. A term is *linear* if it does not contain multiple occurrences of any variable. $\mathcal{LT}(\Sigma, \mathcal{V})$ denotes the set of linear terms.

$t|_p$ denotes the *subterm* of t at position p, and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term s.

Given a substitution $\vartheta = \{x_1/t_1, \dots, x_n/t_n\}$ we denote by $\mathit{dom}(\sigma)$ and $\mathit{range}(\sigma)$ the domain set $\{x_1, \dots, x_n\}$ and the range set $\bigcup_{i=1}^{n} \mathit{var}(t_i)$ respectively. The identity substitution is denoted by ε. By $t\sigma$ we denote the application of σ to t. $\sigma{\restriction}_V$ denotes the restriction of substitution σ to set $V \subseteq \mathcal{V}$. $\sigma\vartheta$ denotes the composition of ϑ and σ i.e., the substitution s.t. $x(\sigma\vartheta) = (x\sigma)\vartheta$ for any $x \in \mathcal{V}$. Given two substitutions ϑ_1 and ϑ_2 and two terms t_1 and t_2, we say that ϑ_1

(respectively t_1) is more general than ϑ_2 (respectively t_2), denoted $\vartheta_1 \leq \vartheta_2$ (respectively $t_1 \leq t_2$) if and only if there exists a substitution σ s.t. $\vartheta_1\sigma = \vartheta_2$ (respectively $t_1\sigma = t_2$). We denote by $\simeq$ the induced equivalence, i.e., $\vartheta \simeq \sigma$ if and only if there exists a renaming ρ s.t. $\vartheta\rho = \sigma$ (and $\sigma\rho^{-1} = \vartheta$). With $\sigma \uparrow \sigma'$ we indicate the *lub* (w.r.t. $\leq$) of σ and σ'.

A substitution ϑ is an *unifier* for terms t and s if $t\vartheta = s\vartheta$. t and s *unify/are unifiable* when there exists a unifier. A unifier σ for t and s is a *most general unifier*, denoted as $\sigma = mgu(t, s)$, when $\sigma \leq \vartheta$ for any unifier ϑ of t and s.

Rewriting

A *term rewriting system* (TRS for short) $\mathcal{R}$ is a set of rules $l \to r$ where $l, r \in \mathcal{T}(\Sigma, \mathcal{V})$, $var(r) \subseteq var(l)$, $l = f(t_1, \dots, t_n)$ and $f \in \mathcal{D}$. $t_1, \dots, t_n$ are the argument patterns of $l \to r$ and need not necessarily be in $\mathcal{T}(\mathcal{C}, \mathcal{V})$, unlike in functional programming, where only *constructor-based* TRSs are considered (i.e., with $t_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$). We denote by $\mathbb{R}_\Sigma$ the set of all TRSs defined on signature Σ.

Given TRS $\mathcal{R}$, a rewrite step $t \xrightarrow[\mathcal{R}]{p} t'$ is defined if there are a position p in t, $l \to r \in \mathcal{R}$ and a substitution η with $dom(\eta) \subseteq var(l)$ such that $t|_p = l\eta$ and $t' = t[r\eta]_p$. As usual, we omit to write position p when it is not relevant and omit $\mathcal{R}$ when is clear from the context. Moreover we use $\to^*$ to denote the transitive and reflexive closure of the rewriting relation $\to$.

A term t is called a *normal form*, denoted by $t \not\to$, if there is no term s such that $t \to^* s$.

A substitution $\{x_1/t_1, \dots, x_n/t_n\}$ is $\mathcal{R}$-normalized (w.r.t. a TRS $\mathcal{R}$) if all t_i are normal forms (which trivially includes the case when $t_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$).

Full Narrowing

In the paper with $s \ll X$ we denote a *renamed apart* variant s of an element belonging to a set of syntactic objects X, i.e., a renaming of variable names of some $x \in X$ that does not contain variables that appear in the context of the definition where $s \ll X$ is used (this is also called "using fresh variables names in s").

The combination of variable instantiation and rewriting is called *narrowing* [7]. Formally, a (full) narrowing step $t \overset{\sigma, p}{\underset{\mathcal{R}}{\leadsto}} t'$ is defined if there is a position p in t, $l \to r \ll \mathcal{R}$ and $\sigma = mgu(t|_p, l)$ such that $t|_p \notin \mathcal{V}$ and $t' = (t[r]_p)\sigma$. In such a case we have that $t\sigma \xrightarrow[\mathcal{R}]{p} t'$. Again, we omit to write position p when it is not relevant and omit $\mathcal{R}$ when is clear from the context.

$t \not\leadsto$ denotes that there is no term s such that $t \leadsto s$.

3 Modeling the small-step rewriting behavior

In this section we introduce the concrete semantics which is suitable to model the small-step rewriting behavior. In order to formally state such relationship we first need to formally define the concept of small-step rewriting behavior.

Definition 3.1 (Rewriting behavior) *Given* $t_0 \in \mathcal{T}(\Sigma, \mathcal{V})$ *and* $\mathcal{R} \in \mathbb{R}_\Sigma$ *the* small-step rewriting behavior of t_0 in $\mathcal{R}$ *is*

$$\mathcal{B}^{ss}[\![t_0 \text{ in } \mathcal{R}]\!] := \left\{ t_0 \xrightarrow[\mathcal{R}]{} t_1 \xrightarrow[\mathcal{R}]{} \cdots \xrightarrow[\mathcal{R}]{} t_{n-1} \xrightarrow[\mathcal{R}]{} t_n \,\middle|\, \forall t_i \in \mathcal{T}(\Sigma, \mathcal{V}) \right. \tag{3.1}$$

and the small-step rewriting behavior of $\mathcal{R}$ *is* $\mathcal{B}^{ss}[\![\mathcal{R}]\!] := \bigcup_{t \in \mathcal{T}(\Sigma, \mathcal{V})} \mathcal{B}^{ss}[\![t \text{ in } \mathcal{R}]\!]$.

This notion of observable behavior induces the definition of TRS equivalence:

$$\forall \mathcal{R}_1, \mathcal{R}_2 \in \mathbb{R}_\Sigma . \mathcal{R}_1 \approx_{ss} \mathcal{R}_2 :\Longleftrightarrow \mathcal{B}^{ss}[\![\mathcal{R}_1]\!] = \mathcal{B}^{ss}[\![\mathcal{R}_2]\!] \tag{3.2}$$

Thanks to the following property we can restrain the check of $\mathcal{B}^{ss}$ equivalence of two TRSs to linear terms. Moreover in the sequel we will also restrict our attention only to denotations for linear terms.

Proposition 3.2 *Let $\mathcal{R}_1, \mathcal{R}_2 \in \mathbb{R}_\Sigma$. Then $\mathcal{R}_1 \approx_{ss} \mathcal{R}_2 \iff \mathcal{R}_1$ is a variant of $\mathcal{R}_2 \iff \forall t \in \mathcal{LT}(\Sigma, \mathcal{V})$. $\mathcal{B}^{ss}[\![t \text{ in } \mathcal{R}_1]\!] = \mathcal{B}^{ss}[\![t \text{ in } \mathcal{R}_2]\!]$.*

3.1 The semantic domain

We now define a notion of "hypothetical rewriting" which resembles full narrowing [7], but it "decouples" multiple occurrences of variables. This way we maintain the potentiality to choose different evolutions of the rewritings of redexes once variables are instantiated with non constructor terms.

The semantic domain of our semantics will be made of trees with all possible derivations of this variation of narrowing.

3.1.1 Linearizing Narrowing

Definition 3.3 (Term Linearization) *Let $t \in \mathcal{T}(\Sigma, \mathcal{V})$, $r \in \mathcal{LT}(\Sigma, \mathcal{V})$ and $\sigma : \mathcal{V} \to \mathcal{V}$. We say that (r, σ) is a* linearization *of t, denoted $(r, \sigma) = lin(t)$, if $r\sigma = t$ and $var(t) \subseteq var(r)$. The substitution σ will be called* delinearizator.

If a term is linear then $lin(t) = (t, \varepsilon)$, while for non-linear terms we can have different possibilities (for instance $lin(f(x,x)) = (f(x,y), \{y/x\}) = (f(z,x), \{z/x\}) = \ldots$). However the following constructions which involve linearization are actually independent upon the particular choice of linearization, in the sense that all possible different results are variants (analogously to what happens for the choice of different *mgu*'s).

It may be useful to note that a delinearizer σ has one binding for each (further) multiple occurrence of a variable.

Definition 3.4 (Linearizing Narrowing Derivation) *Let $t, s \in \mathcal{LT}(\Sigma, \mathcal{V})$ and $\mathcal{R} \in \mathbb{R}_\Sigma$. There exists a* linearizing narrowing step *$t \overset{\theta, p}{\underset{\sigma, \mathcal{R}}{\Longrightarrow}} s$ if there exist a position p of t, $l \to r \ll \mathcal{R}$, $\theta' = mgu(t|_p, l)$ and $\sigma : \mathcal{V} \to \mathcal{V}$ such that*

$$t|_p \notin \mathcal{V}, \qquad (s, \sigma) = lin(t[r\theta']_p), \qquad \theta = \theta'\restriction_{var(t)}.$$

We omit to write position p when it is not relevant and omit $\mathcal{R}$ when is clear from the context.

A sequence $t_0 \overset{\theta_1}{\underset{\sigma_1}{\Longrightarrow}} t_1 \ldots \overset{\theta_n}{\underset{\sigma_n}{\Longrightarrow}} t_n$ is called linearizing narrowing derivation. *With $t_0 \overset{\theta}{\underset{\sigma}{\Longrightarrow}}{}^{*} t_n$ we denote the existence of a linearizing narrowing derivation such that $\theta = \theta_1 \cdots \theta_n$ and $\sigma = \sigma_1 \cdots \sigma_n$.*

Note that in linearizing narrowing derivations we do not apply *mgu* θ to all reduct $(t[r]_p)\theta$ as narrowing does. This would not make any difference since terms are kept linear by construction and thus θ cannot alter the context outside positions being reduced.

Linearizing narrowing is correct w.r.t. rewriting (as narrowing is) as proven by the following theorem which is the analogous of Theorem 3.8 of [8] (Theorem 1 of [7]) where we use linearizing narrowing instead of narrowing. Actually linearizing narrowing is more general than narrowing, in the sense that when we have a narrowing derivation then we will surely have a linearizing narrowing derivation which possibly compute more general instances (in case there are non-linear terms).

Theorem 3.5 *Let $\mathcal{R} \in \mathbb{R}_\Sigma$, $s_0 \in \mathcal{LT}(\Sigma, \mathcal{V})$, $t_0 \in \mathcal{T}(\Sigma, \mathcal{V})$, η_0 an $\mathcal{R}$-normalized substitution such that $t_0 = s_0\eta_0$ and $V \subset \mathcal{V}$ such that $var(s_0) \cup dom(\eta_0) \subseteq V$. If $t_0 \to^* t_n$ then there exist a term $s_n \in \mathcal{LT}(\Sigma, \mathcal{V})$ and substitutions η_n, θ, σ such that*

$$s_0 \overset{\theta}{\underset{\sigma}{\Longrightarrow}}{}^{*} s_n, \qquad t_n = s_n\sigma\eta_n, \qquad (\theta\eta_n)\restriction_V = \eta_0\restriction_V, \qquad \eta_n \textit{ is } \mathcal{R}\textit{-normalized},$$

where $s_0 \overset{\theta}{\underset{\sigma}{\Longrightarrow}}{}^{*} s_n$ and $t_0 \to^* t_n$ employ the same rewrite rules at the same positions.

Note that while it is always possible to transform rewrite derivations into linearizing narrowing derivations (in the sense of Theorem 3.5), the opposite does not hold in general as we will show in Example 3.7. As anticipated, we gather all linearizing narrowing derivations (of the same term t) into a tree.

Definition 3.6 (Linearizing Narrowing Trees) *Let $t \in \mathcal{LT}(\Sigma, \mathcal{V})$. A* linearizing narrowing tree *T for t is a (not necessarily finite) labelled tree which is rooted in t and where*

1. *paths are linearizing narrowing derivations;*
2. *sibling subtrees have the same root terms if and only if their incoming arcs have different substitutions.*

We denote with $\mathbb{LNT}_\Sigma$ (or simply $\mathbb{LNT}$ when clear from the context) the set of all the linearizing narrowing trees (over Σ). Moreover, for any $t \in \mathcal{LT}(\Sigma, \mathcal{V})$, we denote with $\mathbb{LNT}_t$ the set of all linearizing narrowing trees for t.

Point 2 ensures that all sibling steps in a linearizing tree are pairwise distinct and thus that we cannot have two different paths of the tree with the same terms and labels.

Example 3.7

Consider TRS $\mathcal{R}$ of Example 1.1. The linearizing narrowing tree starting from term $\mathit{diff}(x)$ is:

$$\begin{tikzcd}
 & & \mathit{True} \\
\mathit{diff}(x) \arrow[r, Rightarrow, "\varepsilon", "\{x_1/x\}"'] & x \neq x_1 \arrow[ur, Rightarrow, "\{x/\mathit{Tail}, x_1/\mathit{Head}\}", "\varepsilon"'] \arrow[dr, Rightarrow, "\{x/\mathit{Head}, x_1/\mathit{Tail}\}", "\varepsilon"'] & \\
 & & \mathit{True}
\end{tikzcd}$$

This linearizing narrowing derivation $\mathit{diff}(x) \overset{\{x/\mathit{Head}, x_1/\mathit{Tail}\}}{\underset{\{x_1/x\}}{\Longrightarrow}}{}^{*} \mathit{True}$ can be read as: if there is a term t that can rewrite both to Head and Tail, then $\mathit{diff}(t)$ rewrites to True. Indeed $\mathit{diff}(\mathit{coin})$ does rewrite to True (a possible rewriting derivation is $\mathit{diff}(\mathit{coin}) \to \mathit{coin} \neq \mathit{coin} \to \mathit{Head} \neq \mathit{coin} \to \mathit{Head} \neq \mathit{Tail} \to \mathit{True}$).

In this case is not possible to transform these linearizing narrowing derivations into rewrite derivations (since $\mathit{diff}(x) \to x \neq x \not\to$, as well as $\mathit{diff}(t) \to t \neq t \not\to$, for all $t \in \mathcal{T}(\mathcal{C}, \mathcal{V})$).

This example also shows that linearizing narrowing can have longer derivations w.r.t. (standard) narrowing since $\mathit{diff}(x) \overset{\varepsilon}{\leadsto} x \neq x \not\leadsto$.

Definition 3.8 (Variance on $\mathbb{LNT}$) *Let $t \in \mathcal{LT}(\Sigma, \mathcal{V})$ and $T_1, T_2 \in \mathbb{LNT}_t$. We say that T_1 and T_2 are* local variants *if there exists a renaming ρ, such that $T_1\rho = T_2$.*

Two linearizing narrowing trees are local variants if and only if they have the same root t and their steps are equal up to renaming of variables which do not occur in t.

Note: Since the actual choices of local variable names is completely irrelevant, from now on, with an abuse of notation, by $\mathbb{LNT}$ we will actually indicate its quotient w.r.t. local variance. Moreover all linearizing narrowing trees presented in the sequel will actually be an arbitrary representative of an equivalence class.

Definition 3.9 (Order on $\mathbb{LNT}$) *Let denote with $paths(T)$ the set of all the paths of T starting from the root.*

Given $T_1, T_2 \in \mathbb{LNT}$, we define $T_1 \sqsubseteq T_2$ if and only if $paths(T_1) \subseteq paths(T_2)$.

Given a set $\mathcal{T} \subseteq \mathbb{LNT}_t$, the least upper bound $\bigsqcup \mathcal{T}$ is the tree whose paths are $\bigcup_{T \in \mathcal{T}} paths(T)$. Dually for the greatest lower bound $\bigsqcap$.

It is worth noticing that, for any $t \in \mathcal{LT}(\Sigma, \mathcal{V})$, $\mathbb{LNT}_t$ is a complete lattice.

By Point 2 of Definition 3.6, *paths* is injective, thus it establishes an order preserving isomorphism $(\mathbb{LNT}_t, \sqsubseteq) \underset{paths}{\overset{prfxtree}{\leftrightarrows}} (paths(\mathbb{LNT}_t), \subseteq)$, where the adjoint of *paths*, *prfxtree*, builds a tree from a set of paths (by merging all common prefixes). So we have two isomorphic representations of linearizing narrowing trees and in the sequel we can simply write $d \in T$ for $d \in paths(T)$. The set representation is very convenient for technical definitions, while for examples the tree representation is better suited.

Definition 3.10 (Linearizing Narrowing Tree of a term) *Let $t \in \mathcal{LT}(\Sigma, \mathcal{V})$ and $\mathcal{R} \in \mathbb{R}_\Sigma$. A* linearizing narrowing tree *$\mathcal{N}[\![t \text{ in } \mathcal{R}]\!]$ for term t in TRS $\mathcal{R}$ is*

$$\mathcal{N}[\![t \text{ in } \mathcal{R}]\!] := \{d \in \mathbb{LNT}_t | d \text{ uses rules from } \mathcal{R}$$

Intuitively, $\mathcal{N}[\![t \text{ in } \mathcal{R}]\!]$ denotes the linearizing narrowing behavior of linear term t in TRS $\mathcal{R}$ modulo local variance (i.e., local variables are up to renaming).

Example 3.11

Given the following TRS $\mathcal{R}$:

$$\begin{aligned} m(H(x)) &\to d(x, K(x)) & \qquad d(C(x), K(E(y))) &\to f(x, x, y, y) \\ f(A, x, y, F) &\to B(y, y) & \qquad f(E(x), y, A, A) &\to K(E(y)) \end{aligned}$$

The linearizing narrowing tree $\mathcal{N}[\![m(x) \text{ in } \mathcal{R}]\!]$ is:

$$\begin{tikzcd}
 & & & K(E(x_4)) \\
m(x) \arrow[r, Rightarrow, "\{x/H(x_1)\}", "\{x_2/x_1\}"'] & d(x_1, K(x_2)) \arrow[r, Rightarrow, "{\{x_1/C(x_3), x_2/E(y)\}}", "{\{x_4/x_3, y_1/y\}}"'] & f(x_3, x_4, y, y_1) \arrow[ur, Rightarrow, "{\{x_3/E(x_5), y/A, y_1/A\}}", "\varepsilon"'] \arrow[dr, Rightarrow, "{\{x_3/A, y_1/F\}}", "\{y_2/y\}"'] & \\
 & & & B(y, y_2)
\end{tikzcd}$$

Linearizing narrowing trees "capture" the behavioral TRS equivalence $\approx_{ss}$ since the TRS equivalence induced by $\mathcal{N}$ coincides with $\approx_{ss}$.

Theorem 3.12 *Let $\mathcal{R}_1, \mathcal{R}_2 \in \mathbb{R}_\Sigma$. Then $\mathcal{R}_1 \approx_{ss} \mathcal{R}_2$ if and only if, for every $t \in \mathcal{LT}(\Sigma, \mathcal{V})$, $\mathcal{N}[\![t \text{ in } \mathcal{R}_1]\!] = \mathcal{N}[\![t \text{ in } \mathcal{R}_2]\!]$.*

Even more, from linearizing narrowing trees we can completely reconstruct the small-step rewriting behavior. To formally prove this we need to introduce the following operation to build rewriting derivations from linearizing narrowing derivations.

Definition 3.13 *Let $\mathcal{R} \in \mathbb{R}_\Sigma$ and $d = s_0 \overset{\theta_1}{\underset{\sigma_1}{\Longrightarrow}} s_1 \overset{\theta_2}{\underset{\sigma_2}{\Longrightarrow}} \dots \overset{\theta_n}{\underset{\sigma_n}{\Longrightarrow}} s_n$ be a linearizing narrowing derivation. The* linear narrowing to rewriting operator *is defined as*

$$\lfloor d \rfloor := \left\{ t_0 \to t_1 \dots \to t_k \;\middle|\; \begin{array}{l} \xi \textit{ is an } \mathcal{R}\textit{-normalized substitution},\ 0 \le k \le n, \\ \eta = \theta_1 \uparrow \sigma_1 \uparrow \dots \uparrow \theta_k \uparrow \sigma_k,\ \forall 0 \le i \le k.\, t_i = s_i \eta \xi \end{array} \right\} \tag{3.3}$$

We abuse notation and lift $\lfloor \cdot \rfloor$ also to sets as $\lfloor S \rfloor := \bigcup_{d \in S} \lfloor d \rfloor$.

Intuitively, this operation takes a prefix d of a linear narrowing derivation and, if it can simultaneously satisfy all its computed answers and delinearizators, with substitution η, then it builds a rewriting sequence by applying $\eta\xi$ to all terms of d, for any $\mathcal{R}$-normalized substitution ξ.

Theorem 3.14 *Let $\mathcal{R} \in \mathbb{R}_\Sigma$ and $t \in \mathcal{LT}(\Sigma, \mathcal{V})$. Then $\mathcal{B}^{ss}[\![t \textit{ in } \mathcal{R}]\!] = \lfloor \mathcal{N}[\![t \textit{ in } \mathcal{R}]\!] \rfloor$.*

Hence $\mathcal{N}[\![t \textit{ in } \mathcal{R}]\!]$ is indeed a condensed representation of $\mathcal{B}^{ss}[\![t \textit{ in } \mathcal{R}]\!]$. Now the next step for the construction of a semantics with the desired characteristics is to achieve compositionality. To do so we should look for a denotation for most general terms $f(\overrightarrow{x_n})$ (of a TRS $\mathcal{R}$) which could be used to retrieve, with suitable semantic operations, $\mathcal{N}[\![t \textit{ in } \mathcal{R}]\!]$ for any $t \in \mathcal{LT}(\Sigma, \mathcal{V})$.

3.2 Operational denotations of TRSs

The operational denotation of a TRS can be defined as an interpretation giving meaning to the defined symbols over linearizing narrowing trees "modulo variance". Essentially we define the semantics of each function in $\mathcal{D}$ over formal parameters (whose names are actually irrelevant).

Definition 3.15 (Interpretations) *Let $\mathbb{MGT}_\mathcal{D} := \{ f(\overrightarrow{x_n}) \mid f_{/n} \in \mathcal{D}, \overrightarrow{x_n}$ are distinct variables$\}$.*

Two functions $I, J \colon \mathbb{MGT}_\mathcal{D} \to \mathbb{LNT}_\Sigma$ are (global) variants*, denoted by $I \cong J$, if for each $\pi \in \mathbb{MGT}_\mathcal{D}$ there exists a renaming ρ such that $(I(\pi))\rho = J(\pi\rho)$.*

An interpretation *is a function $\mathcal{I} \colon \mathbb{MGT}_\mathcal{D} \to \mathbb{LNT}_\Sigma$ modulo variance*[2] *such that, for every $\pi \in \mathbb{MGT}_\mathcal{D}$, $\mathcal{I}(\pi)$ is a linearizing narrowing tree for π.*

The semantic domain $\mathbb{I}_\Sigma$ (or simply $\mathbb{I}$ when clear from the context) is the set of all interpretations ordered by the pointwise extension of $\sqsubseteq$.

The partial order on $\mathbb{I}$ formalizes the evolution of the computation process. $(\mathbb{I}, \sqsubseteq)$ is a complete lattice and its least upper bound and greatest lower bound are the pointwise extension of $\bigsqcup$ and $\bigsqcap$, respectively. In the sequel we abuse the notations for $\mathbb{LNT}$ for $\mathbb{I}$ as well. The bottom element of $\mathbb{I}$ is $\bot_\mathbb{I} := \lambda\pi.\{\pi\}$ (for each $\pi \in \mathbb{MGT}_\mathcal{D}$). In the sequel we abuse the notations for $\mathbb{LNT}$ for $\mathbb{I}$ as well.

It is important to note that $\mathbb{MGT}_\mathcal{D}$ (modulo variance) has the same cardinality of $\mathcal{D}$ (and is then finite) and thus each interpretation is a finite collection (of possibly infinite elements). Hence we will often explicitly write interpretations by cases, like

$$\mathcal{I} := \begin{cases} \pi_1 \mapsto T_1 \\ \vdots \\ \pi_n \mapsto T_n \end{cases} \qquad \text{for} \qquad \begin{array}{c} \mathcal{I}(\pi_1) := T_1 \\ \vdots \\ \mathcal{I}(\pi_n) := T_n \end{array}$$

[2]i.e., a family of elements of $\mathbb{LNT}_\Sigma$, indexed by $\mathbb{MGT}_\mathcal{D}$, modulo variance.

In the following, any $\mathcal{I} \in \mathbb{I}$ is implicitly considered as an arbitrary function $\mathbb{MGT} \to \mathbb{LNT}$ obtained by choosing an arbitrary representative of the elements of $\mathcal{I}$ in the equivalence class generated by $\cong$. Actually, in the sequel, all the operators that we use on $\mathbb{I}$ are also independent of the choice of the representative. Therefore, we can define any operator on $\mathbb{I}$ in terms of its counterpart defined on functions $\mathbb{MGT} \to \mathbb{LNT}$.

Moreover, we also implicitly assume that the application of an interpretation $\mathcal{I}$ to a specific $\pi \in \mathbb{MGT}$, denoted by $\mathcal{I}(\pi)$, is the application $I(\pi)$ of any representative I of $\mathcal{I}$ which is defined exactly on π. For example if $\mathcal{I} = (\lambda f(x,y).\, f(x,y) \xRightarrow[\varepsilon]{\{x/c(z)\}} c(y,z))/_{\cong}$ then $\mathcal{I}(f(u,v)) = f(u,v) \xRightarrow[\varepsilon]{\{u/c(z)\}} c(v,z)$.

While defined symbols have to be interpreted according to TRS rules, constructor symbols are meant to be interpreted as themselves. In order to treat them as a generic case of function application, we assume that *any* interpretation $\mathcal{I}$ is also implicitly extended on constructors as $\mathcal{I}(c(\overrightarrow{x_n})) \coloneqq c(\overrightarrow{x_n})$. In the sequel we will use φ when we refer to a generic (either constructor or defined) symbol, whence f for defined symbols and c for constructor ones.

Definition 3.16 (Operational denotation of TRSs) *Let* $\mathcal{R} \in \mathbb{R}_\Sigma$. *Then the* operational denotation *of* $\mathcal{R}$ *is*

$$\mathcal{O}[\![\mathcal{R}]\!] \coloneqq \big(\lambda f(\overrightarrow{x_n}).\, \mathcal{N}[\![f(\overrightarrow{x_n}) \textit{ in } \mathcal{R}]\!]\big)/_{\cong} \tag{3.4}$$

Intuitively, $\mathcal{O}$ collects the linearizing narrowing tree of each $f(\overrightarrow{x_n})$ in $\mathcal{R}$, abstracting from the particular choice of the variable names $\overrightarrow{x_n}$.

Example 3.17

The operational denotation of TRS $\mathcal{R}$ of Example 1.1 is:

$$\mathcal{O}[\![\mathcal{R}]\!] = \left\{ \begin{tikzcd}
 & & & Tail \\
coin \mapsto & coin \arrow[urr, Rightarrow, "\varepsilon", "\varepsilon"'] \arrow[drr, Rightarrow, "\varepsilon", "\varepsilon"'] & & \\
 & & & Head \\
 & & & True \\
x \neq y \mapsto & x \neq y \arrow[urr, Rightarrow, "{\{x/Tail, y/Head\}}", "\varepsilon"'] \arrow[drr, Rightarrow, "{\{x/Head, y/Tail\}}", "\varepsilon"'] & & \\
 & & & True \\
 & & & & True \\
diff(x) \mapsto & diff(x) \arrow[r, Rightarrow, "\varepsilon", "{\{x_1/x\}}"'] & x \neq x_1 \arrow[urr, Rightarrow, "{\{x/Head, x_1/Tail\}}", "\varepsilon"'] \arrow[drr, Rightarrow, "{\{x/Tail, x_1/Head\}}", "\varepsilon"'] & & \\
 & & & & True
\end{tikzcd} \right.$$

The (small-step) rewriting behavior of any term t can be "reconstructed" from $\mathcal{O}[\![\mathcal{R}]\!]$ by means of the following *evaluation function* $\mathcal{E}$.

3.2.1 Evaluation Function

When we have the interpretation with the linearizing narrowing tree of $f(\overrightarrow{x_n})$, we can easily reconstruct the rewriting behavior of any $f(\overrightarrow{t_n})$ for (renamed apart) $\overrightarrow{t_n} \in \mathcal{LT}(\mathcal{C},\mathcal{V})$, by simply replacing the most general bindings along a derivation with constructor terms t_i (clearly pruning branches with inconsistent instances). However, with non-constructor nested terms things gets more involved. In practice we have an interleaving of parts of all sub-derivations corresponding to the evaluation of arguments, leaded by the derivation of $f(\overrightarrow{x_n})$. Intuitively the basic building block of our proposal is the definition of a semantics embedding operation that mimics parameter passing. Namely taken two linearizing narrowing trees T_1, T_2 and a variable x of (the root of) T_1, the tree-embedding operation $T_1[x/T_2]$ transforms T_1 by modifying its steps accordingly to steps of T_2, which provides specific actual parameter values to x in places where x in T_1 was originally "freely" instantiated.

In order to define $T_1[x/T_2]$ we need to introduce an auxiliary (technical) relation which works on single derivations. Note that in the sequel, to shorten definitions, *when* we adorn linearizing narrowing derivations or trees with a term s, like d_s or T_s, we mean that the head of d_s or the root of T_s is term s.

Definition 3.18 *Let $\pi \in \mathcal{LT}(\Sigma, \mathcal{V})$, d_g be a linearizing narrowing derivation with head g and $T_b \in \mathbb{LNT}_b$. Then $d_g; \pi; T_b \vdash d$ is the least relation that satisfies the rules:*

$$\frac{}{d_t; \pi; T_s \vdash t\mu} \ \mu = \overleftarrow{mgu}(s, \pi) \tag{3.5a}$$

$$\frac{d_t; \pi; T_{s'} \vdash d}{d_t; \pi; T_s \vdash t\mu \overset{\theta, q}{\underset{\sigma}{\Longrightarrow}} d} \ \mu = \overleftarrow{mgu}(s, \pi)\ ,\ s \overset{\theta, p}{\underset{\sigma}{\Longrightarrow}} T_{s'} \in T_s,\ \exists q\,.\, s|_p = t\mu|_q \tag{3.5b}$$

$$\frac{d_{t_0}; \pi_0; T_{s_0} \vdash d_{t_1} \ \dots \ d_{t_n}; \pi_n; T_{s_n} \vdash d_{t_{n+1}}}{(t \overset{\theta, p}{\underset{\sigma}{\Longrightarrow}} d_{t_0}); \pi; T_{s_0} \vdash t\mu \overset{\theta', p}{\underset{\sigma''}{\Longrightarrow}} d_{t_{n+1}}} \ \begin{array}{l} \mu = \overleftarrow{mgu}(s_0, \pi), \\ \theta' = \overleftarrow{mgu}(t\mu, t\theta)\restriction_{var(t\mu)}\ , \\ \{x_1/y_1, \dots, x_n/y_n\} = proj(var(\pi\theta), \sigma), \\ T_{s_1}, \dots, T_{s_n} \ll T_{s_0}, \\ \pi_0 = \pi\theta,\ \forall i \in \{1, \dots, n\}\ \pi_i = \pi_0\{y_i/x_i\}, \\ \forall j \in \{0, \dots, n\}\ \overrightarrow{v_j} = \overrightarrow{var}(s_j), \\ \sigma'' = (\sigma \cup \bigcup_{i=1}^{n} \overrightarrow{v_i}/\overrightarrow{v_0})\restriction_{var(t_{n+1})} \end{array} \tag{3.5c}$$

where

- *$proj(V, \sigma) = \{x/y \in \sigma \mid y \in V\}$,*
- *$\overleftarrow{mgu}(t, s)$ is an mgu θ of t and s such that $\forall x \in var(t)\ x\theta \notin \mathcal{V}$.*

Broadly speaking, the role of π in a statement $d_t; \pi; T_s \vdash d$ is that of the "parameter pattern" responsible to constrain "freely" instantiated formal parameters in d_t to the actual parameters values which are actually "coming" from T_s. More specifically, Rules 3.5 govern the inlaying of the steps of a linearizing narrowing tree T_s into a derivation d_t. In particular

- The axiom 3.5a stops any further possible inlaying.
- The rule 3.5b considers the case when we want to proceed with an inner linearizing narrowing step employing a step coming from T_s. In this case t plays the role of context and the inner step is done accordingly to the one chosen from T_s. Note that is it possible to do the step only if exists q such that $s|_p = t\mu|_q$. Namely the defined symbol, over which the step is done, needs to be "visible" in $t\mu$.
- The rule 3.5c considers the case when we want to do an outermost step. First, the step has to be compatible with the way we instantiated t so far, namely exists $mgu(t\mu, t\theta)$. Then, we choose from σ only the delinearizers that depend on the variable from which we started the embedding. Each of these delinearizer comes from the linearization of a multiple occurrence of a same variable z. Thus, for each rename z' of z we sequentially embed into z' a (possibly different) sub-derivation coming from a renamed apart variant of

T_s. Note that if we would not have multiple occurrences of z (i.e., $proj(var(\pi\theta),\sigma) = \varnothing$), the rule would simply be

$$\frac{d_{t_0};\pi_0;T_{s_0} \vdash d_{t_1}}{(t \overset{\theta,p}{\underset{\sigma}{\Longrightarrow}} d_{t_0});\pi;T_{s_0} \vdash t\mu \overset{\theta',p}{\underset{\sigma}{\Longrightarrow}} d_{t_1}} \quad \begin{array}{l}\mu = \overleftarrow{mgu}(s_0,\pi)\ ,\\ \theta' = \overleftarrow{mgu}(t\mu,t\theta)\restriction_{var(t\mu)}\ ,\\ \pi_0 = \pi\theta\end{array}$$

Note that in Rules 3.5 we use a selected form of *mgu* (i.e., $\overleftarrow{mgu}$ which does not rename variables in the left argument) in order to avoid to change variable names along the way.

Example 3.19

Consider $\mathcal{O}[\![\mathcal{R}]\!]$ of Example 3.17. Let $T_{coin} := \mathcal{O}[\![\mathcal{R}]\!](coin)$ and $d_0 := diff(x) \overset{\varepsilon}{\underset{\{x_1/x\}}{\Longrightarrow}} d_1 \in \mathcal{O}[\![\mathcal{R}]\!](diff(x))$, where $d_1 = x \neq x_1 \overset{\{x/Head,x_1/Tail\}}{\underset{\varepsilon}{\Longrightarrow}} True$.

Let us build a proof tree to find a derivation d such that $d_0;x;T_{coin} \vdash d$. We have to start with an application of rule (3.5c) which (in this case) has two premises since the delinearizator $\{x_1/x\}$ in the first step of d_0 has one binding. The first subtree which embeds the *Head* branch of T_{coin} into x is

$$\dfrac{\dfrac{\dfrac{}{True;Head;Head \vdash True}\,(3.5a)}{d_1;x;Head \vdash Head \neq x_1 \overset{\{x_1/Tail\}}{\underset{\varepsilon}{\Longrightarrow}} True}\,(3.5c)}{d_1;x;T_{coin} \vdash \underbrace{coin \neq x_1 \overset{\varepsilon}{\underset{\varepsilon}{\Longrightarrow}} Head \neq x_1 \overset{\{x_1/Tail\}}{\underset{\varepsilon}{\Longrightarrow}} True}_{d_3}}\,(3.5b) \qquad \text{(PT)}$$

Now we can build the full proof tree (by building the second subtree which, starting from d_3, can finish to embed the *Tail* branch of T_{coin} into x_1).

$$\dfrac{(PT) \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{True;Tail;Tail \vdash True}\,(3.5a)}{d_2;x_1;Tail \vdash Head \neq Tail \overset{\varepsilon}{\underset{\varepsilon}{\Longrightarrow}} True}\,(3.5c)}{d_2;x_1;T_{coin} \vdash Head \neq coin \overset{\varepsilon}{\underset{\varepsilon}{\Longrightarrow}} Head \neq Tail \overset{\varepsilon}{\underset{\varepsilon}{\Longrightarrow}} True}\,(3.5b)}{d_3;x_1;T_{coin} \vdash coin \neq coin \overset{\varepsilon}{\underset{\varepsilon}{\Longrightarrow}} Head \neq coin \overset{\varepsilon}{\underset{\varepsilon}{\Longrightarrow}} Head \neq Tail \overset{\varepsilon}{\underset{\varepsilon}{\Longrightarrow}} True}\,(3.5c)}{d;x;T_{coin} \vdash diff(coin) \overset{\varepsilon}{\underset{\varepsilon}{\Longrightarrow}} coin \neq coin \overset{\varepsilon}{\underset{\varepsilon}{\Longrightarrow}} Head \neq coin \overset{\varepsilon}{\underset{\varepsilon}{\Longrightarrow}} Head \neq Tail \overset{\varepsilon}{\underset{\varepsilon}{\Longrightarrow}} True}\,(3.5c)$$

We have analogous proof tree for $d;x;T_{coin} \vdash diff(coin) \overset{\varepsilon}{\underset{\varepsilon}{\Longrightarrow}} coin \neq coin \overset{\varepsilon}{\underset{\varepsilon}{\Longrightarrow}} Tail \neq coin \overset{\varepsilon}{\underset{\varepsilon}{\Longrightarrow}} Tail \neq Head \overset{\varepsilon}{\underset{\varepsilon}{\Longrightarrow}} True$. In total we have ten possible proof trees, four of whom that have derivations which reach *True*.

Tree-embedding is simply defined by collecting all contributes of $d;x;T \vdash d'$.

Definition 3.20 (Tree-embedding) *Let $T_g \in \mathbb{LNT}_g$ and $T_b \in \mathbb{LNT}_g$ such that*

1. *T_g and T_b do not share any local variable;*
2. *x is a variable which does not occur in T_b.*

Then the tree-embedding *operation $T_g[x/T_b]$ is defined as $T_g[x/T_b] := \{d \mid d_g \in T_g, d_g;x;T_b \vdash d$.*

The evaluation function is obtained by repeated application of tree-embedding.

Definition 3.21 (Evaluation Function) *Let $t \in \mathcal{LT}(\Sigma, \mathcal{V})$ and $\mathcal{I} \in \mathbb{I}$. The evaluation of t w.r.t. $\mathcal{I}$, denoted $\mathcal{E}[\![t]\!]_{\mathcal{I}}$, is defined by induction on the structure of t as follows:*

$$\mathcal{E}[\![x]\!]_{\mathcal{I}} \coloneqq x \tag{3.6a}$$

$$\mathcal{E}[\![\varphi(\overrightarrow{t_n})]\!]_{\mathcal{I}} \coloneqq \mathcal{I}(\varphi(\overrightarrow{x_n}))[x_1/\mathcal{E}[\![t_1]\!]_{\mathcal{I}}]\dots[x_n/\mathcal{E}[\![t_n]\!]_{\mathcal{I}}] \qquad \overrightarrow{x_n} \text{ renamed apart distinct} \tag{3.6b}$$

Example 3.22

Consider $\mathcal{O}[\![\mathcal{R}]\!]$ of Example 3.17. The evaluation of $\mathcal{E}[\![\mathit{diff}(\mathit{coin})]\!]_{\mathcal{O}[\![\mathcal{R}]\!]}$ is

$$\begin{aligned}
&\mathcal{E}[\![\mathit{diff}(\mathit{coin})]\!]_{\mathcal{O}[\![\mathcal{R}]\!]} = && [\text{by Equation (3.6b) with } n = 1]\\
&\mathcal{O}[\![\mathcal{R}]\!](\mathit{diff}(x))[x/\mathcal{E}[\![\mathit{coin}]\!]_{\mathcal{O}[\![\mathcal{R}]\!]}] = && [\text{by Equation (3.6b) with } n = 0]\\
&\mathcal{O}[\![\mathcal{R}]\!](\mathit{diff}(x))[x/\mathcal{O}[\![\mathcal{R}]\!](\mathit{coin})]
\end{aligned}$$

Then, by completing what shown in Example 3.19, we have that $\mathcal{E}[\![\mathit{diff}(\mathit{coin})]\!]_{\mathcal{O}[\![\mathcal{R}]\!]}$ is

$$\begin{tikzcd}
& \mathit{Tail} \neq \mathit{Tail} & & \mathit{Tail} \neq \mathit{Tail} & \\
& \mathit{diff}(\mathit{Tail}) \arrow[u, Rightarrow, "\varepsilon", "\varepsilon"'] & \mathit{coin} \neq \mathit{Tail} \arrow[ur, Rightarrow, "\varepsilon", "\varepsilon"'] \arrow[r, Rightarrow, "\varepsilon", "\varepsilon"'] & \mathit{Head} \neq \mathit{Tail} \arrow[r, Rightarrow, "\varepsilon", "\varepsilon"'] & \mathit{True} \\
& & \mathit{coin} \neq \mathit{Head} \arrow[r, Rightarrow, "\varepsilon", "\varepsilon"'] \arrow[dr, Rightarrow, "\varepsilon", "\varepsilon"'] & \mathit{Tail} \neq \mathit{Head} \arrow[r, Rightarrow, "\varepsilon", "\varepsilon"'] & \mathit{True} \\
\mathit{diff}(\mathit{coin}) \arrow[r, Rightarrow, "\varepsilon", "\varepsilon"'] \arrow[uur, Rightarrow, "\varepsilon", "\varepsilon"'] \arrow[ddr, Rightarrow, "\varepsilon", "\varepsilon"'] & \mathit{coin} \neq \mathit{coin} \arrow[uur, Rightarrow, "\varepsilon", "\varepsilon"'] \arrow[ur, Rightarrow, "\varepsilon", "\varepsilon"'] \arrow[dr, Rightarrow, "\varepsilon", "\varepsilon"'] \arrow[dddr, Rightarrow, "\varepsilon", "\varepsilon"'] & & \mathit{Head} \neq \mathit{Head} & \\
& & \mathit{Tail} \neq \mathit{coin} \arrow[r, Rightarrow, "\varepsilon", "\varepsilon"'] \arrow[dr, Rightarrow, "\varepsilon", "\varepsilon"'] & \mathit{Tail} \neq \mathit{Tail} & \\
& \mathit{diff}(\mathit{Head}) \arrow[d, Rightarrow, "\varepsilon", "\varepsilon"'] & & \mathit{Tail} \neq \mathit{Head} \arrow[r, Rightarrow, "\varepsilon", "\varepsilon"'] & \mathit{True} \\
& \mathit{Head} \neq \mathit{Head} & \mathit{Head} \neq \mathit{coin} \arrow[r, Rightarrow, "\varepsilon", "\varepsilon"'] \arrow[dr, Rightarrow, "\varepsilon", "\varepsilon"'] & \mathit{Head} \neq \mathit{Tail} \arrow[r, Rightarrow, "\varepsilon", "\varepsilon"'] & \mathit{True} \\
& & & \mathit{Head} \neq \mathit{Head} &
\end{tikzcd}$$

3.2.2 Properties of the TRS operational denotation

The following result states formally that from $\mathcal{O}[\![\mathcal{R}]\!]$ the evaluation function $\mathcal{E}$ can reconstruct the linearizing narrowing tree of any linear term.

Theorem 3.23 *For all $\mathcal{R} \in \mathbb{R}_\Sigma$ and $t \in \mathcal{LT}(\Sigma, \mathcal{V})$, $\mathcal{E}[\![t]\!]_{\mathcal{O}[\![\mathcal{R}]\!]} = \mathcal{N}[\![t \text{ in } \mathcal{R}]\!]$.*

A straightforward consequence of Theorems 3.23 and 3.12 is

Corollary 3.24 *For all $\mathcal{R}_1, \mathcal{R}_2 \in \mathbb{R}_\Sigma$, $\mathcal{O}[\![\mathcal{R}_1]\!] = \mathcal{O}[\![\mathcal{R}_2]\!]$ if and only if $\mathcal{R}_1 \approx_{ss} \mathcal{R}_2$.*

Thus semantics $\mathcal{O}$ is fully abstract w.r.t. $\approx_{ss}$.

3.3 Fixpoint denotations of TRSs

We will now define a bottom-up goal-independent denotation which is equivalent to $\mathcal{O}$ and thus (by Corollary 3.24) adequate to characterize the small-step behavior for TRSs. It is defined as the fixpoint of an abstract immediate operator over interpretations $\mathcal{P}[\![\mathcal{R}]\!]$. This operator is essentially given in terms of evaluation $\mathcal{E}$ of right hand sides of rules. Namely, given an interpretation $\mathcal{I}$, it essentially consists in:

- building an initial linearizing narrowing step for a most general term according to rules' left hand side;
- applying the evaluation operator $\mathcal{E}$ to the right hand side of the rule over $\mathcal{I}$.

Definition 3.25 *Let $\mathcal{R} \in \mathbb{R}_\Sigma$. $\mathcal{P}[\![\mathcal{R}]\!] : \mathbb{I} \to \mathbb{I}$ is defined, for all $f \in \mathcal{D}$ (of arity n), as*

$$\mathcal{P}[\![\mathcal{R}]\!]_{\mathcal{I}}(f(\overrightarrow{x_n})) := \bigsqcup \left\{ f(\overrightarrow{x_n}) \overset{\theta}{\underset{\sigma}{\Rightarrow}} \mathcal{E}[\![r']\!]_{\mathcal{I}} \;\middle|\; \begin{array}{l} f(\overrightarrow{x_n})\theta \to r \ll \mathcal{R}, \\ (r', \sigma) = lin(r) \end{array} \right. \tag{3.7}$$

Moreover we define our fixpoint semantics as $\mathcal{F}[\![\mathcal{R}]\!] := lfp\, \mathcal{P}[\![\mathcal{R}]\!]$.

$\mathcal{F}[\![\mathcal{R}]\!]$ is well defined since $\mathcal{P}[\![\mathcal{R}]\!]$ is continuous.

Proposition 3.26 *Let $\mathcal{R} \in \mathbb{R}_\Sigma$. Then $\mathcal{P}[\![\mathcal{R}]\!]$ is continuous.*

Example 3.27

Let us consider the (artificial) TRS $\mathcal{R} := \{g \to f(h(a)),\ h(a) \to h(b),\ f(h(b)) \to a\}$ taken from [1], which is neither constructor-based nor confluent. The evaluation of the right hand sides of all rules is $\mathcal{E}[\![f(h(a))]\!]_{\bot_{\mathbb{I}}} = \bot_{\mathbb{I}}(f(x))[x/\mathcal{E}[\![h(a)]\!]_{\bot_{\mathbb{I}}}] = f(x)[x/h(a)] = f(h(a))$; $\mathcal{E}[\![h(b)]\!]_{\bot_{\mathbb{I}}} = h(x)[x/b] = h(b)$ and $\mathcal{E}[\![a]\!]_{\bot_{\mathbb{I}}} = a$. Hence

$$\mathcal{I}_1 := \mathcal{P}[\![\mathcal{R}]\!]{\uparrow}1 = \begin{cases} g \mapsto\ g \xRightarrow[\varepsilon]{\varepsilon} f(h(a)) \\ h(x) \mapsto\ h(x) \xRightarrow[\varepsilon]{\{x/a\}} h(b) \\ f(x) \mapsto\ f(x) \xRightarrow[\varepsilon]{\{x/h(b)\}} a \end{cases}$$

In the next iteration we have to evaluate $\mathcal{E}[\![f(h(a))]\!]_{\mathcal{I}_1} = \mathcal{I}_1(f(x))[x/\mathcal{E}[\![h(a)]\!]_{\mathcal{I}_1}]$. Since $\mathcal{E}[\![h(a)]\!]_{\mathcal{I}_1} = \mathcal{I}_1(h(x))[x/\mathcal{E}[\![a]\!]_{\mathcal{I}_1}] = h(a) \overset{\varepsilon}{\underset{\varepsilon}{\Rightarrow}} h(b)$ we have

$$\mathcal{P}[\![\mathcal{R}]\!]{\uparrow}2 = \begin{cases} g \mapsto\ g \xRightarrow[\varepsilon]{\varepsilon} f(h(a)) \xRightarrow[\varepsilon]{\varepsilon} f(h(b)) \xRightarrow[\varepsilon]{\varepsilon} a \\ h(x) \mapsto\ h(x) \xRightarrow[\varepsilon]{\{x/a\}} h(b) \\ f(x) \mapsto\ f(x) \xRightarrow[\varepsilon]{\{x/h(b)\}} a \end{cases}$$

Now, since $\mathcal{P}[\![\mathcal{R}]\!]{\uparrow}3 = \mathcal{P}[\![\mathcal{R}]\!]{\uparrow}2$, this is also the fixpoint $\mathcal{F}[\![\mathcal{R}]\!]$.

Example 3.28

Consider TRS $\mathcal{R}$ of Example 1.1. The iterates of $\mathcal{P}[\![\mathcal{R}]\!]$ are

$$\mathcal{P}[\![\mathcal{R}]\!]{\uparrow}1 = \left\{ \begin{tikzcd} & & \mathit{Tail} \\ \mathit{coin} \mapsto & \mathit{coin} \arrow[ur, Rightarrow, "\varepsilon", "\varepsilon"'] \arrow[dr, Rightarrow, "\varepsilon", "\varepsilon"'] & \\ & & \mathit{Head} \\ & & \mathit{True} \\ x \neq y \mapsto & x \neq y \arrow[ur, Rightarrow, "{\{x/\mathit{Tail}, y/\mathit{Head}\}}", "\varepsilon"'] \arrow[dr, Rightarrow, "{\{x/\mathit{Head}, y/\mathit{Tail}\}}", "\varepsilon"'] & \\ & & \mathit{True} \\ \mathit{diff}(x) \mapsto & \mathit{diff}(x) \arrow[r, Rightarrow, "\varepsilon", "{\{x_1/x\}}"'] & x \neq x_1 \end{tikzcd} \right.$$

$$\mathcal{P}[\![\mathcal{R}]\!]{\uparrow}2 = \begin{cases}
\begin{tikzcd} coin \mapsto coin \arrow[r, Rightarrow, "\varepsilon", "\varepsilon"'] \arrow[dr, Rightarrow, "\varepsilon", "\varepsilon"'] & Tail \\ & Head \end{tikzcd} \\
\begin{tikzcd} x \neq y \mapsto x \neq y \arrow[r, Rightarrow, "\{x/Tail,\, y/Head\}", "\varepsilon"'] \arrow[dr, Rightarrow, "\{x/Head,\, y/Tail\}", "\varepsilon"'] & True \\ & True \end{tikzcd} \\
\begin{tikzcd} diff(x) \mapsto diff(x) \arrow[r, Rightarrow, "\varepsilon", "\{x_1/x\}"'] & x \neq x_1 \arrow[r, Rightarrow, "\{x/Head,\, x_1/Tail\}", "\varepsilon"'] \arrow[dr, Rightarrow, "\{x/Tail,\, x_1/Head\}", "\varepsilon"'] & True \\ & & True \end{tikzcd}
\end{cases}$$

Now, since $\mathcal{P}[\![\mathcal{R}]\!]{\uparrow}3 = \mathcal{P}[\![\mathcal{R}]\!]{\uparrow}2$, this is also the fixpoint $\mathcal{F}[\![\mathcal{R}]\!]$.

Example 3.29

Let us consider the TRS $\mathcal{R} := \{zeros \to {:}(0, zeros),\ take(0, x) \to nil,\ take(s(n), {:}(y, z)) \to {:}(y, take(n, z)),\ f(n) \to take(n, zeros)\}$. The first two iterates of $\mathcal{P}[\![\mathcal{R}]\!]$ are

$$\mathcal{P}[\![\mathcal{R}]\!]{\uparrow}1 = \begin{cases}
\begin{tikzcd} zeros \mapsto zeros \arrow[r, Rightarrow, "\varepsilon", "\varepsilon"'] & {:}(0, \boxed{zeros}) \end{tikzcd} \\
\begin{tikzcd} take(x_1, y_1) \mapsto take(x_1, y_1) \arrow[d, Rightarrow, "\{x_1/0\}", "\varepsilon"'] \arrow[dr, Rightarrow, "{\{x_1/s(x_0),\, y_1/{:}(y_0, z)\}}", "\varepsilon"'] & \\ nil & {:}(y_0, \boxed{take(x_0, z)}) \end{tikzcd} \\
\begin{tikzcd} f(x_0) \mapsto f(x_0) \arrow[r, Rightarrow, "\varepsilon", "\varepsilon"'] & \boxed{take(x_0, zeros)} \end{tikzcd}
\end{cases}$$

$$\mathcal{P}[\![\mathcal{R}]\!]{\uparrow}2 = \begin{cases}
\begin{tikzcd} zeros \mapsto zeros \arrow[r, Rightarrow, "\varepsilon", "\varepsilon"'] & {:}(0, zeros) \arrow[r, Rightarrow, "\varepsilon", "\varepsilon"'] & \boxed{{:}(0, {:}(0, zeros))} \end{tikzcd} \\
\begin{tikzcd} take(x_2, y_2) \mapsto take(x_2, y_2) \arrow[d, Rightarrow, "\{x_2/0\}", "\varepsilon"'] \arrow[dr, Rightarrow, "{\{x_2/s(x_1),\, y_2/{:}(y_1, z_1)\}}", "\varepsilon"'] & & \\ nil & {:}(y_1, take(x_1, z_1)) \arrow[dl, Rightarrow, "\varepsilon", "\varepsilon"'] \arrow[dr, Rightarrow, "{\{x_1/s(x_0),\, z_1/{:}(y_0, z_0)\}}", "\varepsilon"'] & \\ {:}(y_1, nil) & & {:}(y_1, {:}(y_0, take(x_0, z_0))) \end{tikzcd} \\
\begin{tikzcd} f(x_1) \mapsto f(x_1) \arrow[d, Rightarrow, "\varepsilon", "\varepsilon"'] & & \\ take(x_1, zeros) \arrow[d, Rightarrow, "\{x_1/0\}", "\varepsilon"'] \arrow[dr, Rightarrow, "\varepsilon", "\varepsilon"'] & & \\ nil & take(x_1, {:}(0, zeros)) \arrow[dl, Rightarrow, "\{x_1/0\}", "\varepsilon"'] \arrow[dr, Rightarrow, "\{x_1/s(x_0)\}", "\varepsilon"'] & \\ nil & & {:}(0, take(x_0, zeros)) \end{tikzcd}
\end{cases}$$

Note that, to highlight the construction order of the various subtrees, we used indices in variables names that respect the order of introduction and boxed the subtrees which correspond to the previous iterate. By continuing the computation of the iterates, we obtain

$$\mathcal{F}[\![\mathcal{R}]\!] = \begin{cases}
\mathit{zeros} \mapsto \begin{tikzcd} \mathit{zeros} \arrow[r, Rightarrow, "\varepsilon", "\varepsilon"'] & {:}(0, \mathit{zeros}) \arrow[r, Rightarrow, "\varepsilon", "\varepsilon"'] & {:}(0, {:}(0, \mathit{zeros})) \cdots \end{tikzcd} \\
\mathit{take}(x,y) \mapsto \begin{tikzcd}[column sep=small]
\mathit{take}(x,y) \arrow[d, Rightarrow, "\{0/x\}", "\varepsilon"'] \arrow[dr, Rightarrow, "{\{x/s(x'),\, y/{:}(y',z)\}}", "\varepsilon"'] & & \\
\mathit{nil} & {:}(y', \mathit{take}(x',z)) \arrow[dl, Rightarrow, "\varepsilon", "\varepsilon"'] \arrow[dr, Rightarrow, "{\{x'/s(x''),\, z/{:}(y'',z')\}}", "\varepsilon"'] & \\
{:}(y', \mathit{nil}) & & {:}(y', {:}(y'', \mathit{take}(x'',z')))
\end{tikzcd} \\
f(x) \mapsto \begin{tikzcd}[column sep=small]
& f(x) \arrow[d, Rightarrow, "\varepsilon", "\varepsilon"'] & \\
& \mathit{take}(x, \mathit{zeros}) \arrow[dl, Rightarrow, "\{x/0\}"', "\varepsilon"] \arrow[d, Rightarrow, "\varepsilon", "\varepsilon"'] & \\
\mathit{nil} & \mathit{take}(x, {:}(0, \mathit{zeros})) \arrow[dl, Rightarrow, "\{x/0\}"', "\varepsilon"] \arrow[d, Rightarrow, "\varepsilon", "\varepsilon"'] \arrow[dr, Rightarrow, "\{x/s(x')\}", "\varepsilon"'] & \\
\mathit{nil} & \mathit{take}(x, {:}(0, {:}(0, \mathit{zeros}))) & {:}(0, \mathit{take}(x', \mathit{zeros})) \arrow[dl, Rightarrow, "\{x'/0\}"', "\varepsilon"] \arrow[d, Rightarrow, "\{x'/s(x'')\}", "\varepsilon"'] \\
& {:}(0, \mathit{nil}) & {:}(0, \mathit{take}(x'', {:}(0, \mathit{zeros})))
\end{tikzcd}
\end{cases}$$

We can observe that, since terms in $\mathcal{R}$ are linear, the linearizing narrowing trees have just ε as delinearizers and actually they are isomorphic to full narrowing trees.

3.3.1 Properties of the TRS fixpoint denotation

The top-down goal-dependent denotation $\mathcal{O}$ and the bottom-up goal-independent denotation $\mathcal{F}$ are actually equivalent.

Theorem 3.30 (Equivalence of denotations) *Let* $\mathcal{R} \in \mathbb{R}_\Sigma$. *Then* $\mathcal{O}[\![\mathcal{R}]\!] = \mathcal{F}[\![\mathcal{R}]\!]$.

A straightforward consequence of Theorem 3.30 and Corollary 3.24 is

Corollary 3.31 (Correctness and full abstraction of $\mathcal{F}$) *Let* $\mathcal{R}_1, \mathcal{R}_2 \in \mathbb{R}_\Sigma$. *Then* $\mathcal{F}[\![\mathcal{R}_1]\!] = \mathcal{F}[\![\mathcal{R}_2]\!]$ *if and only if* $\mathcal{R}_1 \approx_{ss} \mathcal{R}_2$.

4 Conclusions

We have presented a condensed compositional bottom-up semantics for the *full class* of term rewriting systems which is fully abstract w.r.t. the *small-step* behavior of rewriting.

We are going to use this semantics to define, by abstraction, a condensed bottom-up semantics for the *full class* of term rewriting systems which is fully abstract w.r.t. the *big-step* behavior of rewriting and thus suitable for semantic-based program manipulation tools. Actually we have developed a first proposal of such a semantics (by following this approach) but restricted to the class of left-linear TRSs. We already used it to develop a semantics-based automatic specification synthesis prototype [6] which is giving promising results.

However, we believe that our notion of linearized narrowing which, differently from narrowing, represents faithfully the small-step behavior of rewriting (in a compact way), could be interesting for other applications as well.

References

[1] M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and J. Iborra. A Compact Fixpoint Semantics for Term Rewriting Systems. *Theoretical Computer Science*, 411(37):3348–3371, 2010.

[2] M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract Diagnosis of Functional Programs. In M. Leuschel, editor, *Logic Based Program Synthesis and Transformation – 12th International Workshop, LOPSTR 2002, Revised Selected Papers*, volume 2664 of *Lecture Notes in Computer Science*, pages 1–16, Berlin, 2003. Springer-Verlag.

[3] G. Bacci. *An Abstract Interpretation Framework for Semantics and Diagnosis of Lazy Functional-Logic Languages*. PhD thesis, Dipartimento di matematica e Informatica, Università di Udine, 2011.

[4] G. Bacci and M. Comini. A Fully-Abstract Condensed Goal-Independent Bottom-Up Fixpoint Modeling of the Behaviour of First Order Curry. Submitted for Publication., 2012.

[5] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract Diagnosis. *Journal of Logic Programming*, 39(1-3):43–93, 1999.

[6] M. Comini and L. Torella. TRSynth: a Tool for Automatic Inference of Term Equivalence in Left-linear Term Rewriting Systems. In E. Albert and S.-C. Mu, editors, *PEPM '13, Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 67–70. Acm, 2013.

[7] J.-M. Hullot. Canonical Forms and Unification. In *Proceedings of the 5th International Conference on Automated Deduction*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334, Berlin, 1980. Springer-Verlag.

[8] A. Middeldorp and E. Hamoen. Completeness results for basic narrowing. *Applicable Algebra in Engineering, Communication and Computing*, 5:213–253, 1994. 10.1007/BF01190830.

[9] H. R. Nielson and F. Nielson. Infinitary Control Flow Analysis: a Collecting Semantics for Closure Analysis. In *Symposium on Principles of Programming Languages*, pages 332–345, 1997.

[10] TeReSe, editor. *Term Rewriting Systems*. Cambridge University Press, Cambridge, UK, 2003.

A Technical Proofs

Proof of Proposition 3.2.

Point 1 Implication $\Longleftarrow$ is straightforward. We prove $\Longrightarrow$ by reduction to absurd. Suppose that $\mathcal{R}_1 \approx_{ss} \mathcal{R}_2$ and $\mathcal{R}_1$ is not a variant of $\mathcal{R}_2$. Then there is at least one rule which is different in $\mathcal{R}_1$ w.r.t. $\mathcal{R}_2$ (or vice versa). Thus, by Equation (3.1), we can have rewriting steps which employ that rule in $\mathcal{B}^{ss}[\![\mathcal{R}_1]\!]$ which cannot be in $\mathcal{B}^{ss}[\![\mathcal{R}_2]\!]$ which is absurd.

Point 2 Similarly to previous point, but restricting to initial linear terms.

□

In the sequel we use the following results.

Proposition A.1 ([8]) *Suppose we have substitutions θ, ρ, ρ' and sets A, B of variables such that $(B - dom(\theta)) \cup range(\theta) \subseteq A$. If $\rho\restriction_A = \rho'\restriction_A$ then $(\theta\rho)\restriction_A = (\theta\rho')\restriction_A$.*

Proposition A.2 ([8]) *Let $\mathcal{R}$ be a TRS and suppose we have substitutions θ, ρ, ρ' and sets A, B of variables such that the following conditions are satisfied:*

$$\rho\restriction_A \ \mathcal{R}\text{-normalized} \qquad \theta\rho'\restriction_A = \rho\restriction_A \qquad B \subseteq (A - dom(\theta)) \cup range(\theta\restriction_A)$$

Then $\rho'\restriction_B$ is also $\mathcal{R}$-normalized.

Proof of Theorem 3.5. We prove the thesis by induction on the length of the rewriting derivation from t_0 to t_n. The case of length zero is straightforward.

Suppose $t_0 \to t_1 \to \ldots \to t_n$ is a rewriting derivation of length $n > 0$. We may assume, without loss of generality, that $var(l) \cap V = \varnothing$. We have $(s_0\eta_0)|_p = s_0|_p\eta_0 = \tau l$ for some substitution τ with $dom(\tau) \subseteq var(l)$. Let $\mu := \tau \cup \eta_0$. We have $s_0|_p\mu = s_0|_p\eta_0 = l\tau = l\mu$, so $s_0|_p$ and l are unifiable. Let $\theta_1' := mgu(s_0|_p, l)$ and $\theta_1 := \theta_1'\restriction_{var(s_0|_p)}$. Clearly $dom(\theta_1) \cup range(\theta_1) \subseteq var(s_0|_p) \cup var(l)$. Moreover there exists a substitution ψ such that

$$\theta_1\psi = \mu \tag{1}$$

Now let $(s_1, \sigma_1) := lin(s_0[r\theta_1']_p)$. By Definition 3.4

$$s_0 \xRightarrow[\sigma_1,\, l\to r]{\theta_1,\, p} s_1 \tag{2}$$

Let $V_1 := (V - dom(\theta_1)) \cup range(\theta_1) \cup dom(\sigma_1)$ and

$$\eta_1 := (\sigma_1\psi)\restriction_{V_1} \tag{3}$$

Clearly $dom(\eta_1) \subseteq V_1$. We have $var(s_1) = dom(\sigma_1) \cup var(s_0[r\theta_1]_p) \subseteq dom(\sigma_1) \cup var(s_0[l\theta_1]_p) = dom(\sigma_1) \cup var(s_0\theta_1) \subseteq V_1$. Therefore $var(s_1) \cup dom(\theta_1) \subseteq V_1$. By (1) and (3) we have $s_1\sigma_1\eta_1 = s_1\sigma_1\psi = s_0[r\theta_1]_p\psi = s_0[r]_p\theta_1\psi = s_0[r]_p\mu = s_0\mu[r\mu]_p$. Since $V \cap dom(\tau) = \varnothing$, we have

$$\mu\restriction_V = \eta_0\restriction_V. \tag{4}$$

Likewise $\mu\restriction_{var(r)} = \eta_0\restriction_{var(r)}$. Hence $s_0\mu[r\mu]_p = s_0\eta_0[r\tau]_p = t_0[r\tau]_p = t_1$. Thus

$$t_1 = s_1\sigma_1\eta_1. \tag{5}$$

Next we have to verify that $(\theta_1\eta_1)\restriction_V = \eta_0\restriction_V$. By (3) and by Proposition A.1 we have that $(\theta_1\eta_1)\restriction_V = (\theta_1\sigma_1\psi)\restriction_V$. By (1) and (4), $(\theta_1\eta_1)\restriction_V = (\theta_1\sigma_1\psi)\restriction_V$. Since $dom(\sigma_1) \notin V$, we have that $(\theta_1\sigma_1\psi)\restriction_V = (\theta_1\psi)\restriction_V = \mu\restriction_V = \eta_0\restriction_V$. Thus

$$(\theta_1\eta_1)\restriction_V = \eta_0\restriction_V. \tag{6}$$

Now we show that η_1 is $\mathcal{R}$-normalized. Since $dom(\eta_1) \subseteq V_1$, it suffices to show that $\eta_1\restriction_{V_1}$ is $\mathcal{R}$-normalized. Let $B := (V - dom(\theta_1)) \cup range(\theta_1\restriction_V)$. Proposition A.2 yields the normalization of $\eta_1\restriction_B$. Recall that $range(\theta_1) \subseteq var(s_0|_p) \cup var(l)$. Let $x \in range(\theta_1)$; since θ_1 is idempotent, clearly $x \notin dom(\theta_1)$. If $x \in var(s_0|_p) \subseteq V$ then $x \in V - dom(\theta_1) \subseteq B$. If $x \in var(l)$ then $x \in var(l\theta_1) = var(s_0|_p\theta_1)$ thus $x \in range(\theta_1\restriction_V) \subseteq B$. Thus $range(\theta_1) \subseteq B$ and then $V_1 = B \cup dom(\sigma_1)$. By this and (3), since $\eta_1\restriction_B$ is $\mathcal{R}$-normalized, $\eta_1\restriction_{V_1}$ is $\mathcal{R}$-normalized as well.

By inductive hypothesis we have a term s_n and substitutions θ', σ' and η_n such that

$$s_1 \overset{\theta'}{\underset{\sigma'}{\Longrightarrow}}{}^{*} s_n, \tag{7}$$
$$t_n = s_n\sigma'\eta_n, \tag{8}$$
$$(\theta'\eta_n)\restriction_{V_1} = \eta_1\restriction_{V_1}, \tag{9}$$
$$\eta_n \text{ is } \mathcal{R}\text{-normalized.} \tag{10}$$

Moreover, $s_1 \overset{\theta'}{\underset{\sigma',\mathcal{R}}{\Longrightarrow}}{}^{*} s_n$ and $t_1 \underset{\mathcal{R}}{\longrightarrow}{}^{*} t_n$ apply the same rewrite rules at the same positions. Let $\theta := \theta_1\theta'$ and $\sigma = \sigma_1\sigma'$. By (2) and (7) we obtain $s_0 \overset{\theta}{\underset{\sigma}{\Longrightarrow}}{}^{*} s_n$. By construction this narrowing derivation makes use of the same rewriting rules at the same positions as the rewriting derivation $t_0 \underset{\mathcal{R}}{\longrightarrow}{}^{*} t_n$. It remains to show that $(\theta\eta_n)\restriction_V = \eta_0\restriction_V$. By (9) and Proposition A.1, $(\theta_1\theta'\eta_n)\restriction_V = (\theta_1\eta_1)\restriction_V$. Therefore, by (6), $(\theta\eta_n)\restriction_V = (\theta_1\eta_1)\restriction_V = \eta_0\restriction_V$. □

Proof of Theorem 3.12. We prove $\Longrightarrow$ by reduction to absurd. Suppose that $\mathcal{R}_1 \approx_{ss} \mathcal{R}_2$ and $\exists t \in \mathcal{LT}(\Sigma,\mathcal{V})$. $\mathcal{N}[\![t \text{ in } \mathcal{R}_1]\!] \neq \mathcal{N}[\![t \text{ in } \mathcal{R}_2]\!]$. This means that there is at least one derivation which belongs to $\mathcal{N}[\![t \text{ in } \mathcal{R}_1]\!]$ and not to $\mathcal{N}[\![t \text{ in } \mathcal{R}_2]\!]$ (or vice versa). Hence, there is at least a rule which is not in common to the two programs. Thus $\mathcal{R}_1 \not\approx_{ss} \mathcal{R}_2$ which is absurd.

We prove $\Longleftarrow$ by reduction to absurd. Suppose that $\forall t \in \mathcal{LT}(\Sigma,\mathcal{V})$. $\mathcal{N}[\![t \text{ in } \mathcal{R}_1]\!] = \mathcal{N}[\![t \text{ in } \mathcal{R}_2]\!]$ and $\mathcal{R}_1 \not\approx_{ss} \mathcal{R}_2$. Then there is at least one rule which is different in $\mathcal{R}_1$ w.r.t. $\mathcal{R}_2$. Thus, by Equation (3.1) and Theorem 3.5, $\mathcal{N}[\![t \text{ in } \mathcal{R}_1]\!] \neq \mathcal{N}[\![t \text{ in } \mathcal{R}_2]\!]$ which is absurd. □

Proof of Theorem 3.14. We prove the two inclusions separately. Inclusion $\supseteq$ is immediate by Definition 3.13.

Inclusion $\subseteq$ is proved by reduction to absurd. Assume that there exists a derivation $t_0 \underset{\mathcal{R}}{\longrightarrow}{}^{*} t_n$ such that it does not belong to $\lfloor\mathcal{N}[\![t_0 \text{ in } \mathcal{R}]\!]\rfloor$. Then, by Theorem 3.5, taken $\eta_0 = \varepsilon$, there exists a relaxed narrowing derivation of the form $s_0 \overset{\theta_1}{\underset{\sigma_1}{\Longrightarrow}} \dots \overset{\theta_n}{\underset{\sigma_n}{\Longrightarrow}} s_n$. Note that, for each $i \in \{1..n\}$, θ_i and σ_i are just renaming, $\eta_i = \varepsilon$ and $t_i = s_i\sigma_i \dots \sigma_0$. Let $\eta := \eta_0 = \varepsilon$. It is easy to see that there exists η' such that $\eta' = \theta_1 \uparrow \sigma_1 \uparrow \dots \uparrow \theta_n \uparrow \sigma_n$. By Definition 3.13, there exists a rewriting derivation $t'_0 \underset{\mathcal{R}}{\longrightarrow}{}^{*} t'_n$ where $t'_i = s_i\eta'$ for each $i \in \{1..n\}$. Moreover, $dom(\sigma_k) \notin var(s_i)$ for each $k \in \{i+1..n\}$ and $\theta_i = (\sigma_j)\restriction_{dom(\theta_i)}$ or $\theta_i = (\sigma_j)\restriction_{dom(\theta_i)}$ for some j. Hence $s_i\eta' = s_i\sigma_i \dots \sigma_0$. Thus $t'_i = t_i$ for each $i \in \{1..n\}$ which is absurd. □

Lemma A.3 *Let $\mathcal{R}$ a TRS, $x \in \mathcal{V}$, and $t, s \in \mathcal{T}(\Sigma,\mathcal{V})$ such that they do not share variables and $x \in t, x \notin s$. Then, $\mathcal{N}[\![t \text{ in } \mathcal{R}]\!][x/\mathcal{N}[\![s \text{ in } \mathcal{R}]\!]] = \mathcal{N}[\![t\{x/s\} \text{ in } \mathcal{R}]\!]$*

Proof . It is sufficient to prove that, given $d_t \in \mathcal{N}[\![t \text{ in } \mathcal{R}]\!]$ and $T_s = \mathcal{N}[\![s \text{ in } \mathcal{R}]\!]$, if $d_t; x; T_s \vdash d$ then $d \in \mathcal{N}[\![t\{x/s\} \text{ in } \mathcal{R}]\!]$. To this aim we will prove a stronger result: let $\pi \in \mathcal{T}(\mathcal{C},\mathcal{V})$, let $T_s = \mathcal{N}[\![s \text{ in } \mathcal{R}]\!]$, and for any $d_t \in \mathcal{N}[\![t \text{ in } \mathcal{R}]\!]$ such that $d_t; \pi; T_s \vdash d$ then $d \in \mathcal{N}[\![t\eta \text{ in } \mathcal{R}]\!]$ where $\eta = \overleftarrow{mgu}(s,\pi)$. We proceed by structural induction on the proof tree.

rule 3.5a) straightforward.

rule 3.5b) By inductive hypothesis $d_t; \pi; T_{s'} \vdash d_{t\mu} \iff d_{t\mu} \in \mathcal{N}[\![t\mu \text{ in } \mathcal{R}]\!]$. If it exists $d_{t\mu}$, then $t\mu|_p = s|_p$. Moreover, $\exists\mu'$ such that $\mu' = \overleftarrow{mgu}(s',\pi)$ and $t\mu'|_p = s'|_p$. Thus, $t\mu \overset{\theta,p}{\underset{\sigma}{\Longrightarrow}} d_{t\mu} \in \mathcal{N}[\![t\mu \text{ in } \mathcal{R}]\!]$.

rule 3.5c) By inductive hypothesis we have that $d_{t_0}; \pi_0; T_{s_0} \vdash d_{t_1} \iff d_{t_1} \in \mathcal{N}[\![t\mu_0 \text{ in } \mathcal{R}]\!], \dots, d_{t_n}; \pi_n; T_{s_n} \vdash d_{t_{n+1}} \iff d_{t_{n+1}} \in \mathcal{N}[\![t\mu_n \text{ in } \mathcal{R}]\!]$, where $\mu_0 = \overleftarrow{mgu}(s_0, \pi_0), \dots, \mu_n = \overleftarrow{mgu}(s_n, \pi_n)$. We know that exists a linearizing narrowing step $t \overset{\theta,p}{\underset{\sigma}{\Longrightarrow}} t_0$ where $\theta_2 = mgu(t|_p, l), \theta = \theta_2 {\restriction}_{var(t)}, (s, \sigma) = lin(r\theta_2), t\mu = t[s]_p$. We need to prove that it exists the linearizing narrowing step $t\mu \overset{\theta',p}{\underset{\sigma''}{\Longrightarrow}} t_{n+1}$. Since it exists $mgu(t|_p, l)$ and $mgu(t\mu, t\theta)$, then it exists θ_3 such that $\theta_3 = mgu((t\mu)|_p, l)$. Now let $\theta' := \theta_3 {\restriction}_{var(t\mu)}$ and $(s', \sigma') = lin(r\theta_3)$, then $t\mu[s']_p = \bar{t}$. Let us observe that $t_1 = t_0\mu_0, t_2 = t_0\mu_0\mu_1, \dots, t_{n+1} = t_0\mu_0 \dots \mu_n$ where $\mu_0 = \overleftarrow{mgu}(s_0, \pi_0), \dots, \mu_n = \overleftarrow{mgu}(s_n, \pi_n)$. Let $\mu^* = \mu_0 \dots \mu_n$, then $t_{n+1} = t_0\mu^* = t\mu^*[s\mu^*]_p = t\mu[s\mu^*]_p = \bar{t}$. □

□

Proof of Theorem 3.23. We proceed by structural induction on term t

t* = *x Immediate by Equation (3.6a).

***t* = $\boldsymbol{\varphi(\overrightarrow{t_n})}$**

$$\begin{aligned}
&\mathcal{E}[\![\varphi(\overrightarrow{t_n})]\!]_{\mathcal{O}[\![\mathcal{R}]\!]} = \\
&\qquad [\text{by Equation (3.6b)}] \\
&= \mathcal{O}[\![\mathcal{R}]\!](\varphi(\overrightarrow{x_n}))[x_1/\mathcal{E}[\![t_1]\!]_{\mathcal{O}[\![\mathcal{R}]\!]}] \dots [x_n/\mathcal{E}[\![t_n]\!]_{\mathcal{O}[\![\mathcal{R}]\!]}] = \\
&\qquad [\text{by inductive hypothesis}] \\
&= \mathcal{N}[\![\varphi(\overrightarrow{x_n}) \text{ in } \mathcal{R}]\!][x_1/\mathcal{N}[\![t_1 \text{ in } \mathcal{R}]\!]] \dots [x_n/\mathcal{N}[\![t_n \text{ in } \mathcal{R}]\!]] = \\
&\qquad [\text{by Lemma A.3}] \\
&= \mathcal{N}[\![\varphi(\overrightarrow{t_n}) \text{ in } \mathcal{R}]\!]
\end{aligned}$$

□

Proof of Proposition 3.26. It is straightforward to prove that $\mathcal{P}[\![P]\!]$ is monotone and finitary, thus it is continuous. □

Proof of Theorem 3.30. We prove the two inclusions separately.

$\sqsubseteq$) Let indicate with $\mathcal{O}_k$ all derivations of $\mathcal{O}[\![\mathcal{R}]\!]$ with length $\le k$. by induction we prove that $\forall k\ \mathcal{O}_k \sqsubseteq \mathcal{P}[\![\mathcal{R}]\!]{\uparrow}k$.

k = 0) immediate.

k > 0) $\forall d \in \mathcal{O}_k$, we have that $d = f(x) \overset{\theta,\Lambda}{\underset{\sigma}{\Longrightarrow}} d'$ such that (by Theorem 3.23) $d'_t \in \mathcal{E}[\![t]\!]_{\mathcal{O}_{k-1}}$. Thus, by monotonicity of $\mathcal{E}$, we have that $d'_t \in \mathcal{E}[\![t]\!]_{\mathcal{P}[\![\mathcal{R}]\!]{\uparrow}k-1}$. By the definition of $\mathcal{P}$, $d \in \mathcal{P}[\![P]\!]_{\mathcal{P}[\![\mathcal{R}]\!]{\uparrow}k-1} = \mathcal{P}[\![\mathcal{R}]\!]{\uparrow}k$

Thus, by Proposition 3.26, $\mathcal{O}[\![\mathcal{R}]\!] = \bigsqcup_{k\ge0} \mathcal{O}_k \sqsubseteq \bigsqcup_{k\ge0} \mathcal{P}[\![\mathcal{R}]\!]{\uparrow}k = \mathcal{F}[\![\mathcal{R}]\!]$.

$\sqsupseteq$) We need to prove that $f(\overrightarrow{x_n}) \overset{\theta,\Lambda}{\underset{\sigma}{\Longrightarrow}} \mathcal{E}[\![r']\!]_{\mathcal{O}[\![\mathcal{R}]\!]} \sqsubseteq \mathcal{N}[\![f(\overrightarrow{x_n}) \text{ in } \mathcal{R}]\!]$. By Theorem 3.23, $\mathcal{E}[\![r']\!]_{\mathcal{O}[\![\mathcal{R}]\!]}$ is a linearizing narrowing tree. Since $\theta = mgu(f(\overrightarrow{x_n}, l)){\restriction}_{\overrightarrow{x_n}}$ and $(\sigma, r') = lin(r)$, we can conclude that $f(\overrightarrow{x_n}) \overset{\theta,\Lambda}{\underset{\sigma}{\Longrightarrow}} \mathcal{E}[\![r']\!]_{\mathcal{O}[\![\mathcal{R}]\!]} \sqsubseteq \mathcal{N}[\![f(\overrightarrow{x_n}) \text{ in } \mathcal{R}]\!]$.

□

Logical and Algebraic Views of a Knot Fold of a Regular Heptagon

Fadoua Ghourabi[1], Tetsuo Ida[2] and Kazuko Takahashi[1]

[1] Kwansei Gakuin University, Japan
ghourabi@kwansei.ac.jp, ktka@kwansei.ac.jp
[2] University of Tsukuba, Japan
ida@i-eos.org

Abstract

Making a knot on a rectangular origami or more generally on a tape of a finite length gives rise to a regular polygon. We present an automated algebraic proof that making two knots leads to a regular heptagon. Knot fold is regarded as a double fold operation coupled with Huzita's fold operations. We specify the construction by describing the geometrical constraints on the fold lines to be used for the construction of a knot. The algebraic interpretation of the logical formulas allows us to solve the problem of how to find the fold operations, i.e. to find concrete fold lines. The logical and algebraic framework incorporated in a system called Eos (e-origami system) is used to simulate the knot construction as well as to prove the correctness of the construction based on algebraic proof methods.

1 Introduction

From the early history of mathematics, the correspondence between geometry and algebra has been recognized and has been the subject of constant study. In some sense, ancient geometry was a mother of algebra. In particular, solving algebraic equations has been related to the realm of Euclidean geometry. Early mathematicians, notably Khawarizmi and Khayyam, gave geometrical meanings to the solutions of equations. Khayyam's insight was that solutions of certain cubic equations are points of intersections of conic sections [1]. Seven centuries later, after freeing algebra from geometrical thinking, Wantzel proved that solving cubic solutions is impossible by Euclidean tools, i.e. compass and straightedge [2]. Further tools have been invented and used to perform constructions that are impossible by Euclidean tools.

Paper folding, i.e. origami, allows solving cubic equations and, hence, geometrical constructions such as trisection of an arbitrary angle or a regular heptagon are realizable. Given an origami paper, what we can do by hand is to construct creases and points. The creases are constructed by folding the paper along the lines which we call *fold lines*. The points are constructed by the intersection of lines. Huzita presented a set of fold operations (known also as Huzita's axioms in literature) with which he showed how to obtain fold lines [3]. Huzita's fold operations are simple to perform by hand but powerful enough to solve cubic equations [4].

The geometrical construction of a regular heptagon has a unique history in that it belongs to the famous classical impossible problems by Euclidean tools. Using paper folding, the construction was shown to be possible. In this paper, we show another method of constructing a regular heptagon based on an extension of Huzita's fold operations by introducing knotting. The constructions of regular n-gons (in particular of regular pentagon and heptagon) by making knots have not been rigorously specified, and the observation that their constructions purport to problems of solving geometrical constraints is missing.

We investigate the geometrical constraints of the knot fold towards constructions of regular polygons and proofs of the correctness of these constructions. The knot fold operation is specified by an existentially quantified formula of the first-order predicate logic. The formula is then translated into a set of algebraic equalities and disequalities, which then are solved by specialized solvers based on Gröbner basis theory and/or CAD (Cylindrical Algebraic Decomposition). We have developed a computational origami system called Eos [5] using computer algebra system *Mathematica*. In brief, it is an *e*-origami system which allows us to do *virtual origami*. It has capabilities of constructing and visualizing origami geometrical objects, algebraically analyzing origami folds, and proving the correctness of origami constructions. Eos supports Huzita's fold operations and has been extended to include multi-fold operations. We use Eos, to assist us with our mathematical study of the knot fold.

The rest of the paper is organized as follows. In Sect. 2, we give the notations that we use. In Sect. 3, we present Huzita's fold operations and their extensions. In Sect. 4, we explain the geometrical properties of the knot fold. The construction of regular heptagon by Eos is shown in Sect. 5, and its formal and algebraic proof is discussed in Sect. 6. In Sect. 7, we summarize our results and point out a direction of further research.

2 Notations

In this paper, we restrict the use of geometrical objects to points, segments and lines. Points are denoted by a single capital letter of the Latin alphabet, e.g. A, B, C, D, X, Y etc. Lines are denoted by m, n, t, u and v. Let X and Y be two points, then $\overline{XY}$ denotes the line passing through points X and Y. For brevity, we often write XY to refer to the line passing through X and Y. We use the same notation to denote the segment between points X and Y. Although the meaning of the notation XY should be clear from the context, we precede the notation XY with either the word "segment" or the word "line" to emphasize which objects we are working with. We also use $\overrightarrow{XY}$ to denote a vector from point X to point Y. The distance between two points X and Y is denoted by $|XY|$.

Since we use Cartesian coordinate system in this paper, a line is represented by a linear equation $ax + by + c = 0$ in variables x and y. The sets of all points and lines are denoted by Π and $\mathcal{L}$, respectively. Abusing the set notation, we use $X \in m$ to mean that point X is incident to line m, and $\{X_1, \ldots, X_p\} \subset m$ to mean that all the points $X_1, \ldots, X_p$ are incident to m.

When an origami is folded along a line m, some of the points are moved to superpose with their reflections across m. We denote by X^m the reflection of point X across line m.

A simple knot is denoted by $\mathcal{K}$. $\mathcal{K}_i$ denotes the ith knot. The notation $\mathcal{K} = \langle m, n, t\rangle$ means that the knot $\mathcal{K}$ is defined by 3 fold lines (to be explained in Sect. 4) m, n and t, and furthermore $\mathcal{K}$ is obtained by folding along the lines following the order of their appearance in $\langle m, n, t\rangle$.

3 Origami Geometrical Construction

3.1 Huzita's Fold Operations

By $\mathcal{O}$ we denote an origami. An origami $\mathcal{O}$ is supposed to represent a square sheet of paper with four points on the corners and four edges that is subject to folding[1]. We call A, B, C and D, the points on the corner. Some intersections of lines may not fit on the square paper.

[1]We could take $\mathcal{O}$ to be any convex polygon that can be constructed from a square sheet of paper. However, this could be an unnecessary generalization in our study.

As we want to work with these points, we consider $\mathcal{O}$ to be a sufficiently large (bounded) 2D plane so that all the points and lines of interest are on $\mathcal{O}$. Huzita observed that the degree of freedom of paper fold by fold lines can be made finite by specifying how certain points and lines are superposed. Then, he gave the following operations (O1) $\sim$ (O6), which serve as basic operations in the geometrical construction of origamis. Operation (O7) was added, later, by Justin [6]. We call collectively (O1) $\sim$ (O7) Huzita's fold operations.

(O1) Given two distinct points P and Q, fold $\mathcal{O}$ along the unique line that passes through P and Q.

(O2) Given two distinct points P and Q, fold $\mathcal{O}$ along the unique line to superpose P and Q.

(O3) Given two distinct lines m and n, fold $\mathcal{O}$ along a line to superpose m and n.

(O4) Given a line m and a point P, fold $\mathcal{O}$ along the unique line passing through P to superpose m onto itself.

(O5) Given a line m, a point P not on m and a point Q, fold $\mathcal{O}$ along a line passing through Q to superpose P and m.

(O6) Given two lines m and n, a point P not on m and a point Q not on n, where m and n are distinct or P and Q are distinct, fold $\mathcal{O}$ along a line to superpose P and m, and Q and n.

(O7) Given two lines m and n and a point P not on m, fold $\mathcal{O}$ along the unique line to superpose P and m, and n onto itself.

We note that the above statements are slightly different from the original ones. To formalize the above statements with the view to rigorous geometrical construction, we restate Huzita's operations by carefully adding and removing side conditions of degeneracy and incidence [7]. In essence, treating origamis with a program requires rigorous specification of these operations.

Huzita's fold operations determine fold lines by specifying superpositions of constructed points and lines. In Eos, these specifications are formulas in a language of many-sorted first-order predicate logic. By the algebraic interpretation of the formulas we derive polynomial equalities. The problem of finding fold line(s) is therefore reduced to solving constraints expressed in multi-variate polynomials of degree 3 over the field of origami constructible numbers [8, 5].

3.2 Extensions

The contribution of Huzita's fold operations is powerful enough to perform relevant geometrical constructions by way of solving algebraic equations of degree up to 3. Examples of such constructions are trisecting an arbitrary angle, constructing a regular heptagon, etc. Some studies have explored possible extensions of Huzita's fold operations in an attempt to increase the power of paper folding, i.e. to solve higher degree equations.

Alperin and Lang proposed multi-fold method, where the origami is folded along more than one fold line, simultaneously [9]. The idea is to find fold lines that are mutually dependent. The multi-fold construction allows solving higher degree equations. We presented a construction method of angle trisection using 2-fold operation [5] and angle quintisection using 4-fold operation [10]. Although the p-fold method generates an arbitrarily high degree polynomial, accurately folding an origami by p lines simultaneously would be difficult to do by hand even for $p = 2$.

Folding polygonal knots has been traditionally used in Japan. For example, folding a broad sash for a kimono by a pentagon is a common practice of daily life. However, it is unknown when the knot fold was first mathematically studied. The earliest contributions of which we are aware are those of Cundy and Rollett [11], Brunton [12] and Sakaguchi [13]. Cundy and Rollett showed models of knots that make some regular polygons. Brunton elaborated the mathematical study of the knot fold and showed that the number of necessary knots depends on the number of the edges of the polygon.

4 Knot Fold

In order to determine the geometrical constraints of knot fold, we first analyze key geometrical properties of a knot.

4.1 Geometrical Properties

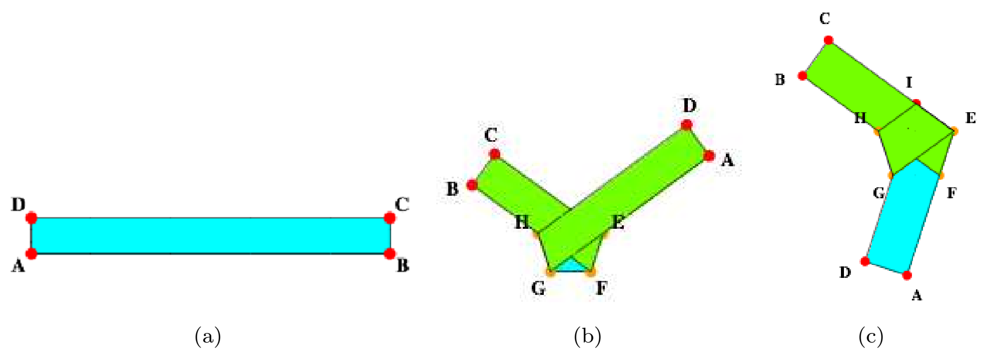


Figure 1: Knot-fold of regular pentagon $FEIHG$

Let us examine the operation of knotting in Fig. 1. We make one simple knot by tying together the two ends of the origami tape in Fig. 1(a). Note that we use a rectangular shape of an origami. When the height of the tape, i.e $|AD|$ and $|BC|$, is infinitesimal and both ends of the paper are connected, the tape becomes a curve, i.e. an object of study in knot theory. The knot with 3 crossings is the most basic one in knot theory. When we bring the height back to the original without distorting the tape, except for folds, we obtain the polygonal knot. A well fastened and well flattened knot becomes a polygonal shape as depicted in Fig. 1(c). The obtained polygonal shape exhibits a regular pentagonal form. As it is inferred from the knot theory, making the knot in Fig. 1(c) requires 3 folds along the lines m, n and t that extend the edges FE, GH and IE^n, respectively.[2] The knot $\mathcal{K} = \langle m, n, t\rangle$ is the one in Fig. 1(c).

When the knot is entirely unfolded, we obtain the tape with the creases and the generated points as shown in Fig. 2. We note the following geometrical properties. The vertices E, H and I are lined up on the edge CD whereas vertices F and G are incident to the edge AB. The fold along m passes through F and superposes point H and line AB. Point E is the intersection of m and CD. Similarly, the fold line n passes through G, and the fold along n superposes point E and line AB. Point H is the intersection of n and CD. Note that fold lines m and n are mutually defined. The fold is a 2-fold operation where m and n are defined simultaneously. Line t can be determined by applying operation (O5) of Huzita's fold operation set. Namely,

[2]Recall that E^n is the reflection point of E across line n as defined in Sect. 2

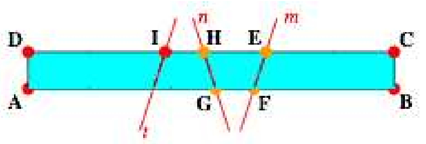


Figure 2: Unknotted tape

line t passes through I and superposes G and CD. Note that the parameters of (O5), points I, G and line CD are considered with respect to the configuration before the fold. Refer to Fig. 2. The origami after the the 2-fold (i.e. along m and n) is shown in Fig. 1(b). The purposes of the fold along t is, first, to construct the vertex I and, second, to fasten and lock the knot.

A purely geometrical proof that the knot fold creates a regular polygon is sketched in [14]. In the following, we present a formal and automated algebraic proof that uses properties of the operation of folding such as preservation of some of distances, parallelism, etc.

4.2 Geometrical Constraint Solving Approach

The question now becomes how to determine the fold lines m, n and t such that the knot is a regular pentagon. The proof that the knot fold gives rise to regular pentagons shows that each of the fold line makes angles $2\alpha = \frac{2\pi}{5}$ with the edges [14], where $\alpha = \measuredangle GEH = \measuredangle EHF$. In origami geometry, construction of angles using Huzita's fold operations is not trivial. The problem of constructing angles is boiled down to a sequence of fold steps which makes the construction tedious where the number of intermediate points and lines is prone to increase. Furthermore, not all angles are constructible by Huzita's fold operations. The construction of a regular 11-gon, whose interior angles are equal to $\frac{9\pi}{11}$, is shown to be impossible by Huzita's fold operations [15]. Hence, we carefully choose a specification of the knot fold that could be extended towards the construction of regular n-gons, where $n \geq 7$. We are led to avoid construction of angles in order to solve the knot fold construction problem. We consider the knot fold construction as a geometrical constraint solving problem without use of angles.

Example: Regular pentagon
Our method of constructing a regular pentagon by the knot fold uses a multi-fold together with Huzita's fold operations. We consider the example in Fig. 1, where we make a knot $\mathcal{K} = \langle m, n, t \rangle$. We try to define the geometrical properties of $\mathcal{K}$ in a systematic way so that we can generalize it to the knot fold of regular n-gons, where $n \geq 7$. We observe the following geometrical properties on m, n, t, E, F, G, H and I.

- Points E and F are incident to m.
- Points G and H are incident to n.
- Points E, H and I are incident to CD.
- Points F and G are incident to AB.
- The fold line m superposes H and AB, i.e. $H^m \in AB$.
- The fold line n superposes E and AB, i.e. $E^n \in AB$.
- The fold line t passes through I and superposes G and CD. i.e. $I \in t$ and $G^t \in CD$.
- The distances $|EF|$, $|FG|$ and $|GH|$ are equal.

The above properties are described by the following first order logical formula $\phi_{\mathcal{K}}$.

$$\begin{aligned}\phi_{\mathcal{K}} \equiv\ & \exists m,n,t \in \mathcal{L}\ \exists E,F,G,H,I \in \Pi \\ & \{E,F\} \subset m \wedge \{G,H\} \subset n \wedge \{E,H,I\} \subset CD \wedge \{F,G\} \subset AB \wedge \\ & H^m \in AB \wedge E^n \in AB \wedge I \in t \wedge G^t \in CD \wedge |EF| = |FG| = |GH|\end{aligned} \tag{1}$$

We used Eos to construct the regular pentagon in Fig. 1. It is possible to determine t independently from m and n. We consider the formula (2), that defines the 2-fold operation to solve lines m and n, first.

$$\begin{aligned}\phi'_{\mathcal{K}} \equiv\ & \exists m,n \in \mathcal{L}\ \exists E,F,G,H \in \Pi \\ & \{E,F\} \subset m \wedge \{G,H\} \subset n \wedge \{E,H\} \subset CD \wedge \{F,G\} \subset AB \wedge \\ & H^m \in AB \wedge E^n \in AB \wedge |EF| = |FG| = |GH|\end{aligned} \tag{2}$$

The fold line t is obtained by applying operation (O5). Namely, line t passes through point I and superposes point H and line AB.

We can prove automatically that the generated shape is a regular pentagon based on Gröbner bases theory. We will omit the proof of the correctness of the pentagon knot construction. In the next sections, we will discuss the construction and the proof of a regular heptagon by the knot fold in details.

5 Regular Heptagon by the Knot Fold

Brunton studied the construction of regular n-gons, where $n \geq 3$ [12]. He showed that the number of necessary knots is $\frac{\Phi(n)}{2} - 1$ in order to construct a regular n-gon, where $\Phi(n)$ is Euler's totient function. Hence, to construct a regular heptagon, we perform 2 knots $\mathcal{K}_1$ and $\mathcal{K}_2$. We explained in Sect. 4 that a knot can be decomposed into 3-fold operations along three fold lines. In the case of regular heptagon, we need 5 fold lines m, n, t, u and v, where $\mathcal{K}_1 = \langle m,n,t \rangle$ and $\mathcal{K}_2 = \langle v,u,m \rangle$. Let $EKJIHGF$ be the constructed regular heptagon. Figure 3 exhibits the vertices of the regular heptagon after unfolding $\mathcal{K}_1$ and $\mathcal{K}_2$. The fold lines are extensions of edges of $EKJIHGF$ as follows. Lines m, n and t are the extension of segment EF, IJ and GF^n, respectively. Lines v and u are extensions of segments KE^u and HI^m, respectively. Figure 4 shows a sketch of a regular heptagon with the lines m, n, t, u and v.

5.1 Geometrical Constraints

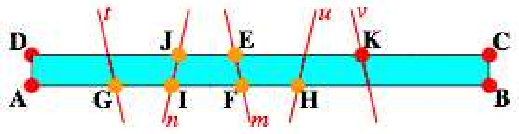


Figure 3: Unfolded regular heptagon EKJIHGF

Lines m and n of the knot $\mathcal{K}_1 = \langle m,n,t \rangle$ are defined by the following properties on m, n, t, E, F, I, J, G and H.

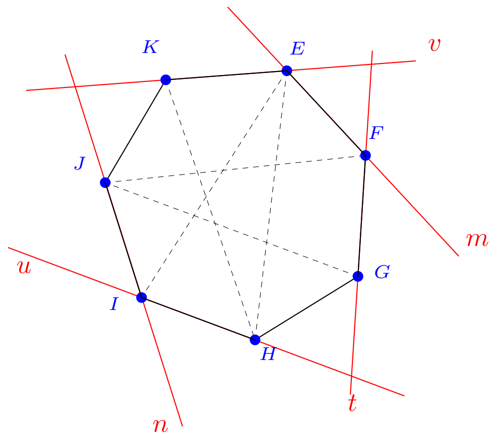


Figure 4: Sketch of a regular heptagon $EKJIHGF$ and lines m, n, t, u and v

- Points E and F are incident to m.
- Points I and J are incident to n.
- Points E and J are incident to CD.
- Points F, I, G and H are incident to AB.
- The fold line m superposes I and CD, i.e. $I^m \in CD$.
- The fold line n superposes F and CD, i.e. $F^n \in CD$.
- The distances $|EF|$, $|IJ|$, $|FG^n|$, $|IH^m|$ and $|G^nH^m|$ are equal.[3]

We write formula $\phi_{\mathcal{K}_1}$ in a similar fashion to the formula (1).

$$\begin{aligned}\phi_{\mathcal{K}_1} \equiv\ & \exists m, n, t \in \mathcal{L}\ \exists E, F, I, J, G, H \in \Pi \\ & \{E, F\} \subset m \wedge \{I, J\} \subset n \wedge \{F, I, G, H\} \subset AB \wedge \{E, J\} \subset CD \wedge \\ & I^m \in CD \wedge F^n \in CD \wedge G \in t \wedge J^t \in AB \wedge \\ & |EF| = |IJ| = |FG^n| = |IH^m| = |G^nH^m|\end{aligned} \tag{3}$$

Similarly to our discussion in Sect. 4.2, line t can be constructed independently from lines m and n. We therefore can separate the construction t and use $\phi'_{\mathcal{K}_1}$ in (4) to solve for m and n, first.

$$\begin{aligned}\phi'_{\mathcal{K}_1} \equiv\ & \exists m, n\ \exists E, F, I, J, G, H \in \Pi \\ & \{E, F\} \subset m \wedge \{I, J\} \subset n \wedge \{F, I, G, H\} \subset AB \wedge \{E, J\} \subset CD \wedge \\ & I^m \in CD \wedge F^n \in CD \wedge \\ & |EF| = |IJ| = |FG^n| = |IH^m| = |G^nH^m|\end{aligned} \tag{4}$$

Knot $\mathcal{K}_2 = \langle v, u, m \rangle$ is defined by the following properties on v, u, m, K, X, H, Y, Z and F.

- Points K and X are on v.
- Points H and Y are on u.
- Points K and Y are on CD.
- Points Z, H, F and X are on AB.
- The fold line v superposes H and CD, i.e. $H^v \in CD$.

[3]Refering to Fig. 3, distances $|GF^n|$ and $|FG^n|$ are equal due to the fact that reflection across line n preserves the distances. Similarly, distances $|HI^m|$ and $|IH^m|$ are equal.

- The fold line u superposes X and CD, i.e. $X^u \in CD$.
- The fold line m passes through F and superposes Y and AB, i.e. $Y^m \in AB$.
- The distances $|KX|$, $|HY|$, $|XF^u|$, $|HZ^v|$ and $|F^uZ^v|$ are equal.

$$\begin{aligned}\phi_{\mathcal{K}_2} \equiv\ & \exists v,u,m \in \mathcal{L}\ \exists K,X,H,Y,Z,F \in \Pi \\ & \{K,X\} \subset v \wedge \{H,Y\} \subset u \wedge \{K,Y\} \subset CD \wedge \{X,H,Z,F\} \subset AB \wedge \\ & H^v \in CD \wedge X^u \in CD \wedge F \in m \wedge Y^m \in AB \wedge \\ & |KX| = |HY| = |XF^u| = |HZ^v| = |F^uZ^v|\end{aligned} \tag{5}$$

However, referring to Fig. 3, points X, Y and Z are equal to points E^u, I^m and $((((G^n)^m)^u)^v)$. From formula $\phi'_{\mathcal{K}_1}$, E, I and G are obtained. It is possible to determine the fold lines that make $\mathcal{K}_2$ in an easier way than solving constraints in formula (5), as we explain in the next sub-section.

5.2 Construction by Eos

We first assume that the initial origami is a rectangle $ABCD$. We work with the Cartesian coordinate system, and the coordinates of points A, B, C and D are (0, 0), (wd, 0), (wd, ht) and (0, ht), respectively. The width wd and the height ht can be taken arbitrary. Of course wd is sufficiently larger than ht, so that the knot construction is feasible. For concreteness of our presentation using Eos, we set $wd = 900$ and $ht = 60$. Furthermore, let E be an arbitrary but fixed point on segment CD. Let the coordinates of E be (400, ht) for simplicity and clarity of the construction. Our objective is to construct a regular heptagon $EKJIHGF$ by the knot fold. Recall that the construction requires a 2-fold operation prior to Huzita's fold operations (see Sect. 4.1). The first folding step is the crucial one, i.e. the 2-fold operation.

In Eos, Huzita's fold operations can be performed using function `HO`. The extension to the multi-fold is natural as `HO` is implemented with the generality that allows the specification of logical formulas describing the multi-fold. Multi-fold is realized by the call of the following *Mathematica* function.

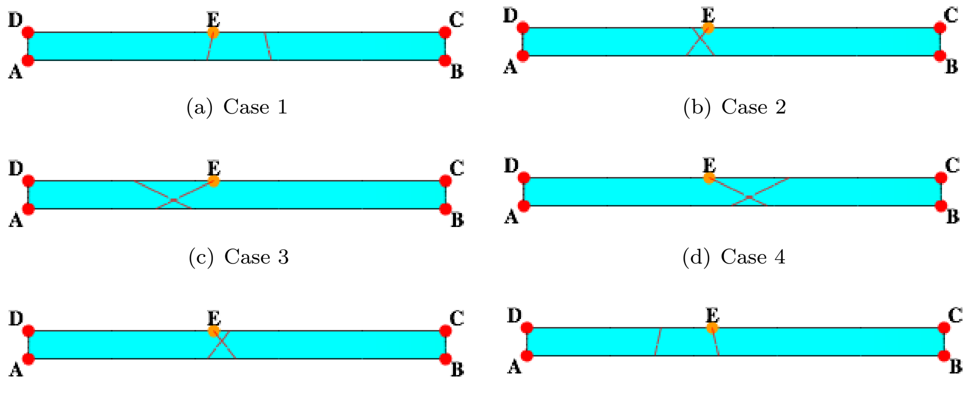
$$\texttt{HO}\,[\ \mathcal{H},\ \texttt{Constraint} \to \phi\ ]$$

$\mathcal{H}$ is a list of points on the origami which determine the faces to be moved. ϕ is a formula in the first-order predicate logic. The formula ϕ specifies the constraints that the geometrical objects concerned have to satisfy. In our example, the geometrical constraints that specify the 2-fold operation are expressed by formula $\phi'_{\mathcal{K}_1}$ in (4). We write $\phi'_{\mathcal{K}_1}$ in the language of Eos and call `HO` as follows.

$$\begin{aligned}&\texttt{HO}[\{\texttt{C, A}\}, \texttt{Constraint} \to \exists_{m,m:\texttt{Line}}\exists_{n,n:\texttt{Line}}\exists_{f,f:\texttt{Point}}\exists_{i,i:\texttt{Point}}\exists_{j,j:\texttt{Point}}\exists_{g,g:\texttt{Point}}\exists_{h,h:\texttt{Point}} \\ &\quad (\{\texttt{E}, f\} \subset m \wedge \{i,j\} \subset n \wedge \{f,i,g,h\} \subset \texttt{AB} \wedge j \in \texttt{CD} \wedge \\ &\quad i^m \in \texttt{CD} \wedge f^n \in \texttt{CD} \wedge \\ &\quad \texttt{SqDistance}[\texttt{E}, f] == \texttt{SqDistance}[i,j] == \texttt{SqDistance}[f, g^n] == \\ &\quad \texttt{SqDistance}[i, h^m] == \texttt{SqDistance}[g^n, h^m])]\end{aligned} \tag{6}$$

The term $\texttt{SqDistance}[X, Y]$ is the square of the distance between X and Y (i.e. $|XY|^2$). Notations like "$\exists_{m,m:\texttt{Line}}$", "$\{\texttt{E}, f\} \subset m$", "$i^m$", "$i^m \in \texttt{CD}$" are Eos extension of Mathematica syntax.

The evaluation of (6) generates 24 distinct cases of possible configurations of points F, G, H, I and J and lines m and n. We deduce that the algebraic interpretation of the constraints, as defined by $\phi'_{\mathcal{K}_1}$, is a well constrained system of equations since we obtained finite number of solutions. However, not all of the 24 cases are relevant to our construction problem. They may



(a) Case 1 (b) Case 2 (c) Case 3 (d) Case 4 (e) Case 5 (f) Case 6

Figure 5: 6 possible cases for lines m and n

correspond to degenerate cases that lead to failure of the proof by Gröbner bases. In order to eliminate the degenerate cases and also to get a reasonable number of cases, we further narrow the search space of the configuration of points and lines. For that, we add the following extra constraints.

$$E \notin n \wedge j \notin m \wedge \texttt{SqDistance}[E,i] == \texttt{SqDistance}[j,f] == \texttt{SqDistance}[E,h] == \texttt{SqDistance}[j,g]$$

The sub-formula $\texttt{E} \notin n \wedge j \notin m$ eliminates the degenerate cases where E and j are equal. Sub-formula $\texttt{SqDistance}[E,i]$ == $\texttt{SqDistance}[j,f]$ == $\texttt{SqDistance}[E,h]$ == $\texttt{SqDistance}[j,g]$ specifies further conditions on points E, F, G, H, I and J. We evaluate the following `HO` call.

$$\begin{aligned}
&\texttt{HO}[\{\texttt{C, A}\}, \texttt{Constraint} \to \exists_{m,m:\texttt{Line}}\exists_{n,n:\texttt{Line}}\exists_{f,f:\texttt{Point}}\exists_{i,i:\texttt{Point}}\exists_{j,j:\texttt{Point}}\exists_{g,g:\texttt{Point}}\exists_{h,h:\texttt{Point}}\\
&(\{\texttt{E}, f\} \subset m \wedge \{i,j\} \subset n \wedge \{f,i,g,h\} \subset \texttt{AB} \wedge j \in \texttt{CD} \wedge\\
&i^m \in \texttt{CD} \wedge f^n \in \texttt{CD} \wedge\\
&\texttt{SqDistance}[\texttt{E}, f] == \texttt{SqDistance}[i,j] == \texttt{SqDistance}[f, g^n] ==\\
&\texttt{SqDistance}[i, h^m] == \texttt{SqDistance}[g^n, h^m] \wedge\\
&\texttt{E} \notin n \wedge j \notin m \wedge \texttt{SqDistance}[E,i] == \texttt{SqDistance}[j,f] == \texttt{SqDistance}[E,h] == \texttt{SqDistance}[j,g]),\\
&\texttt{MarkPointAt} \to \{\texttt{F,I,J,G,H}\}]
\end{aligned} \tag{7}$$

Consequently, we obtain 6 cases as shown in Fig. 5. Two of them, namely those in Fig. 5(a) and in Fig. 5(f), lead to the construction of a regular heptagon. Eos allows the user to interactively choose the suitable case to proceed with the construction. Keyword `MarkPointAt` tells the Eos how to label the existentially quantified points f, i, j, g and h. The outcome of folding in the case 6 is shown in Fig. 6

Next, we apply two operations (O1) to fold along the line FG^n and IH^m in Fig. 6 by calling `HO` with suitable arguments. The lines FG^n and IH^m are the fold lines t and u, respectively. The results are shown in Fig. 7(a) and 7(b). Finally, given the origami of Fig. 7(b), we construct the remaining vertex K. We apply operation (O6) to obtain v, which brings H and G onto IC and HB, respectively. Point K is the intersection of IC and line v. We obtain the heptagon $EKJIHGF$ in Fig. 7(c).

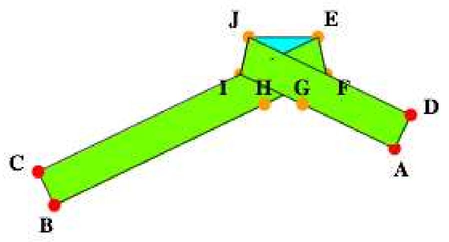


Figure 6: Step 1: Folds along lines m and n

5.3 Algebraic Interpretation

This algebraic interpretation is used to “evaluate” the function calls of `HO` as well as the automated proof of the correctness of the construction. We take `HO` in (7) as an example. To obtain geometrical objects m, n, F, G, H, I and J, the formula in (7) is transformed into a set of algebraic equalities. Further details of the algebraic interpretation are explained in [8] and [5].

Let the coordinates of A, B, D and E be (0, 0), (900, 0), (0, 60) and (400, 60), respectively. For an atomic formula ϕ, let $[\![\phi]\!]$ denote the set of polynomial relations that are the algebraic meaning of ϕ. An atomic formula is interpreted as a set of polynomial relations (equalities or inequalities), and a term is given as a rational function. The set of (non-simplified) polynomial equalities (8) – (14) is the algebraic interpretation of formula in (7).

$$\begin{aligned}\{&400\mathrm{a5} + 60\mathrm{b6} + \mathrm{c7} == 0, \mathrm{c7} + \mathrm{a5x11} + \mathrm{b6y12} == 0, \mathrm{c10} + \mathrm{a8x13} + \mathrm{b9y14} == 0,\\ &\mathrm{c10} + \mathrm{a8x15} + \mathrm{b9y16} == 0, -900\mathrm{y12} == 0, -900\mathrm{y14} == 0,\\ &-900\mathrm{y20} == 0, -900\mathrm{y18} == 0, -54000 + 900\mathrm{y16} == 0\end{aligned} \tag{8}$$

$$-54000 + \frac{900\left(-2\mathrm{b6}(\mathrm{c7} + \mathrm{a5x13}) + \mathrm{a5}^2\mathrm{y14} - \mathrm{b6}^2\mathrm{y14}\right)}{\mathrm{a5}^2 + \mathrm{b6}^2} == 0, \tag{9}$$

$$-54000 + \frac{900\left(-2\mathrm{b9}(\mathrm{c10} + \mathrm{a8x11}) + \mathrm{a8}^2\mathrm{y12} - \mathrm{b9}^2\mathrm{y12}\right)}{\mathrm{a8}^2 + \mathrm{b9}^2} == 0, \tag{10}$$

$$\begin{aligned}&(400 - \mathrm{x11})^2 + (60 - \mathrm{y12})^2 == (\mathrm{x13} - \mathrm{x15})^2 + (\mathrm{y14} - \mathrm{y16})^2 ==\\ &\left(\mathrm{y12} - \frac{-2\mathrm{b9}(\mathrm{c10} + \mathrm{a8x17}) + \mathrm{a8}^2\mathrm{y18} - \mathrm{b9}^2\mathrm{y18}}{\mathrm{a8}^2 + \mathrm{b9}^2}\right)^2 +\\ &\left(\mathrm{x11} - \frac{-\mathrm{a8}^2\mathrm{x17} + \mathrm{b9}^2\mathrm{x17} - 2\mathrm{a8}(\mathrm{c10} + \mathrm{b9y18})}{\mathrm{a8}^2 + \mathrm{b9}^2}\right)^2 ==\\ &\left(\mathrm{y14} - \frac{-2\mathrm{b6}(\mathrm{c7} + \mathrm{a5x19}) + \mathrm{a5}^2\mathrm{y20} - \mathrm{b6}^2\mathrm{y20}}{\mathrm{a5}^2 + \mathrm{b6}^2}\right)^2 +\\ &\left(\mathrm{x13} - \frac{-\mathrm{a5}^2\mathrm{x19} + \mathrm{b6}^2\mathrm{x19} - 2\mathrm{a5}(\mathrm{c7} + \mathrm{b6y20})}{\mathrm{a5}^2 + \mathrm{b6}^2}\right)^2 ==\\ &\left(\frac{-2\mathrm{b9}(\mathrm{c10} + \mathrm{a8x17}) + \mathrm{a8}^2\mathrm{y18} - \mathrm{b9}^2\mathrm{y18}}{\mathrm{a8}^2 + \mathrm{b9}^2} - \frac{-2\mathrm{b6}(\mathrm{c7} + \mathrm{a5x19}) + \mathrm{a5}^2\mathrm{y20} - \mathrm{b6}^2\mathrm{y20}}{\mathrm{a5}^2 + \mathrm{b6}^2}\right)^2 +\\ &\left(\frac{-\mathrm{a8}^2\mathrm{x17} + \mathrm{b9}^2\mathrm{x17} - 2\mathrm{a8}(\mathrm{c10} + \mathrm{b9y18})}{\mathrm{a8}^2 + \mathrm{b9}^2} - \frac{-\mathrm{a5}^2\mathrm{x19} + \mathrm{b6}^2\mathrm{x19} - 2\mathrm{a5}(\mathrm{c7} + \mathrm{b6y20})}{\mathrm{a5}^2 + \mathrm{b6}^2}\right)^2,\end{aligned} \tag{11}$$

$$400\mathrm{a8} + 60\mathrm{b9} + \mathrm{c10} \neq 0, \mathrm{c7} + \mathrm{a5x15} + \mathrm{b6y16} \neq 0, \quad (12)$$

$$(-400 + \mathrm{x13})^2 + (-60 + \mathrm{y14})^2 == (\mathrm{x11} - \mathrm{x15})^2 + (\mathrm{y12} - \mathrm{y16})^2 == (-400 + \mathrm{x19})^2 + (-60 + \mathrm{y20})^2 == (\mathrm{x15} - \mathrm{x17})^2 + (\mathrm{y16} - \mathrm{y18})^2, \quad (13)$$

$$(-1 + \mathrm{b9})\mathrm{b9} == 0, (-1 + \mathrm{a8})(-1 + \mathrm{b9}) == 0, 1 + \mathrm{a8}^2 \neq 0, (-1 + \mathrm{b6})\mathrm{b6} == 0$$
$$(-1 + \mathrm{a5})(-1 + \mathrm{b6}) == 0, 1 + \mathrm{a5}^2 \neq 0\} \quad (14)$$

A line $a\ x + b\ y + c = 0$ is represented by (a, b, c), together with the constraint $(-1 + b)b = 0 \wedge (-1 + a)(-1 + b) = 0 \wedge a^2 + 1 \neq 0$. Lines m and n are represented by $(a5, b6, c7)$ and $(a8, b9, c10)$, respectively. Hence, we have the equalities and disequalities in (14).

The equalities in (8) are the algebraic interpretation of the sub-formula $\{E, f\} \subset m \wedge \{i, j\} \subset n \wedge \{f, i, g, h\} \subset AB \wedge j \in CD$. The first equation $400a5 + 60b6 + c7 == 0$ means that point E at (400, 60) is incident to m defined by the equation $a5x + b6y + c7 = 0$. Similarly, the rest of the equations in (8) are interpretations of f at ($x11$, $y12$) is on m, i at ($x13$, $y14$) is on n, j at ($x15$, $y16$) is on n, f is on AB, i is on AB, h at ($x19$, $y20$) is on AB, g at ($x17$, $y18$) is on AB and j is on CD, respectively.

The reflection of point i at ($x13$, $y14$) across line m is the point i^m whose coordinates are

$$\left(\frac{-a5^2x13 + b6^2x13 - 2a5(c7 + b6y14)}{a5^2 + b6^2}, \frac{-2b6(c7 + a5x13) + a5^2y14 - b6^2y14}{a5^2 + b6^2}\right)$$

Equation (9) states that i^m is incident to CD represented by (0, 1, -60). Similarly, equation (10) is the algebraic interpretation of $f^n \in CD$. Eos transforms rational form $\frac{p}{q} == 0$ to $p == 0$. Note that in (9)–(11), q comes from coefficient of lines and $q \neq 0$ is deduced from (14).

The disequalities of (12) states that the coordinates of point E and j do not satisfy the equation of lines n and m, respectively. Eos changes the disequalities into equalities by adding slack variables introduced by Rabinowitch trick.

Now, we examine the equalities (11) and (13). $[\![\texttt{SqDistance}[E, f]\texttt{==}\texttt{SqDistance}[i, j]]\!]$ gives rise to the first equality in (11), namely $(400-x11)^2+(60-y12)^2 == (x13-x15)^2+(y14-y16)^2$, where E at (400, 60), f at ($x11$, $y12$), i at ($x13$, $y14$) and j at ($x15$, $y16$). The rest of the equalities in (11) and equalities in (13) are obtained in the same way.

By solving the above set of polynomial equalities for the coefficients of m and n and coordinates of f, i, j, g and h, we obtain the 6 cases in Fig. 5.

6 Proof

6.1 Theorem to Prove

We prove the following theorem.

Theorem 6.1. *Given the origami in Fig. 7(c), we have*
(a) $|EF| = |FG| = |GH| = |HI| = |IJ| = |JK| = |KE|$, *and*
(b) $\measuredangle EOF = \measuredangle FOG = \measuredangle GOH = \measuredangle HOI = \measuredangle IOJ = \measuredangle JOK = \measuredangle KOE = \frac{2\pi}{7}$, *where O is the center of $EKJIHGF$.*

Let $\theta = \measuredangle EOF$ and $\alpha = \mathrm{e}^{i\theta}$. Vector $\overrightarrow{FG}$ is the rotation of $\overrightarrow{EF}$ through θ around center O. To prove the theorem 6.1, we show that the rotations of $\overrightarrow{EF}$ through angles 2θ, 3θ, 4θ, 5θ, and

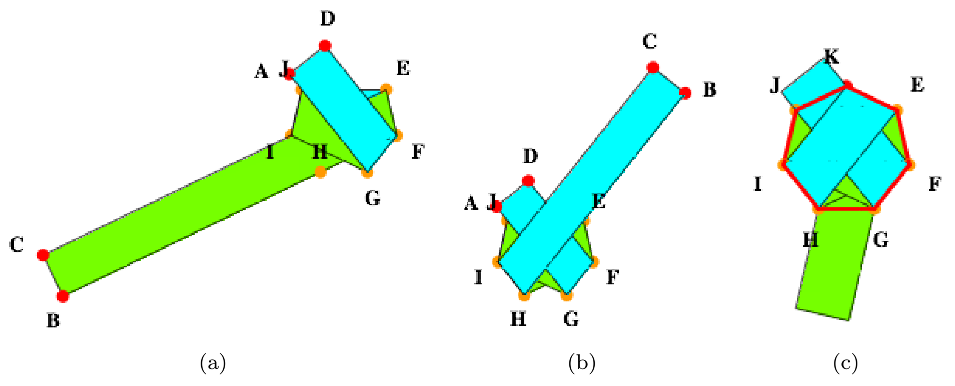


Figure 7: Construction of (a) the edge GF (step 2), (b) the edge IH (step 3) and (c) vertex K and final heptagon $EKJIHG$ (step 4)

6θ gives $\overrightarrow{GH}$, $\overrightarrow{HI}$, $\overrightarrow{IJ}$, $\overrightarrow{JK}$ and $\overrightarrow{KE}$, respectively, and furthermore, that $\theta = \frac{2\pi}{7}$. We show that after the construction, the following holds.

$$\begin{aligned}\forall \alpha \in \mathbb{C} \quad (\alpha\overrightarrow{EF} - \overrightarrow{FG} = 0 \Rightarrow \\ \alpha^2\overrightarrow{EF} - \overrightarrow{GH} = 0 \wedge \alpha^3\overrightarrow{EF} - \overrightarrow{HI} = 0 \wedge \alpha^4\overrightarrow{EF} - \overrightarrow{IJ} = 0 \wedge \\ \alpha^5\overrightarrow{EF} - \overrightarrow{JK} = 0 \wedge \alpha^6\overrightarrow{EF} - \overrightarrow{KE} = 0 \wedge \alpha^7 - 1 = 0)\end{aligned} \tag{15}$$

Let $\mathcal{P}$ be the geometrical constraints accumulated during the construction. Let $\mathcal{C} \equiv \forall\alpha \in \mathbb{C}\ (\mathcal{C}_1 \Rightarrow \mathcal{C}_2 \wedge \mathcal{C}_3 \wedge \mathcal{C}_4 \wedge \mathcal{C}_5 \wedge \mathcal{C}_6 \wedge \mathcal{C}_7)$ be the formula (15). $\mathcal{P}$ and $\mathcal{C}$ form the premise and conclusion of the proposition (16) that we want to prove.

$$\mathcal{P} \Rightarrow \forall\alpha \in \mathbb{C} \quad (\mathcal{C}_1 \Rightarrow \mathcal{C}_2 \wedge \mathcal{C}_3 \wedge \mathcal{C}_4 \wedge \mathcal{C}_5 \wedge \mathcal{C}_6 \wedge \mathcal{C}_7) \tag{16}$$

Formula (16) is semantically equivalent to the conjunctions of the formulas (17)–(22).

$$\mathcal{P} \Rightarrow \forall\alpha \in \mathbb{C}\ (\mathcal{C}_1 \Rightarrow \mathcal{C}_2) \tag{17}$$
$$\mathcal{P} \Rightarrow \forall\alpha \in \mathbb{C}\ (\mathcal{C}_1 \Rightarrow \mathcal{C}_3) \tag{18}$$
$$\mathcal{P} \Rightarrow \forall\alpha \in \mathbb{C}\ (\mathcal{C}_1 \Rightarrow \mathcal{C}_4) \tag{19}$$
$$\mathcal{P} \Rightarrow \forall\alpha \in \mathbb{C}\ (\mathcal{C}_1 \Rightarrow \mathcal{C}_5) \tag{20}$$
$$\mathcal{P} \Rightarrow \forall\alpha \in \mathbb{C}\ (\mathcal{C}_1 \Rightarrow \mathcal{C}_6) \tag{21}$$
$$\mathcal{P} \Rightarrow \forall\alpha \in \mathbb{C}\ (\mathcal{C}_1 \Rightarrow \mathcal{C}_7) \tag{22}$$

6.2 Proof by Eos

We show how the validity of (17) is proved and the rest is achieved in a similar way. We add the conclusion $\forall\alpha \in \mathbb{C}\ \mathcal{C}_1 \Rightarrow \mathcal{C}_2$ by calling Eos function `Goal` as follows.

$$\begin{aligned}\texttt{Goal}[\forall_{\alpha,\alpha\in\texttt{Complexes}}(\texttt{VectorToComplex}[\alpha\overrightarrow{EF} - \overrightarrow{FG}]\texttt{==}0 \Rightarrow \\ (\texttt{VectorToComplex}[\alpha^2\overrightarrow{EF} - \overrightarrow{GH}]\texttt{==}0))]\end{aligned} \tag{23}$$

Expressions $\mathcal{C}_1 \equiv \alpha\overrightarrow{EF} - \overrightarrow{FG} = 0$ and $\mathcal{C}_2 \equiv \alpha^2\overrightarrow{EF} - \overrightarrow{GH} = 0$ written in the language of Eos are `VectorToComplex`$[\alpha\overrightarrow{EF} - \overrightarrow{FG}]$ and `VectorToComplex`$[\alpha^2\overrightarrow{EF} - \overrightarrow{GH}]$`==`0, respectively.

Function `Goal` adds the negation of the conclusion to the premise. By calling `Goal`, we obtain $\mathcal{P} \wedge \neg(\forall \alpha \in \mathbb{C}\ (\mathcal{C}_1 \Rightarrow \mathcal{C}_2))$. In order to translate $\mathcal{P} \wedge \neg(\forall \alpha \in \mathbb{C}\ (\mathcal{C}_1 \Rightarrow \mathcal{C}_2))$ into algebraic form, we fix the coordinate system to be Cartesian with points A, B, C, D and E as follows:

$$\begin{aligned}\texttt{map} = \texttt{DefMapping}[\{\{\texttt{A, Point[0,-wd]}\},\{\texttt{B, Point[wd,0]}\},\\ \{\texttt{C}, \texttt{Point}[\texttt{wd}, \texttt{ht}]\}, \{\texttt{D}, \texttt{Point}[0, \texttt{ht}]\}, \{\texttt{E}, \texttt{Point}[0, \texttt{ht}]\}\}, \{\}]\&\end{aligned} \tag{24}$$

Without loss of generality, we set the size of the initial origami to be $2wd \times ht$, where the width wd is taken to be arbitrary and the height ht is equal to 1. The point E is fixed at location $(0,\ ht)$. Finally, we check whether the reduced Gröbner basis of the algebraic interpretation of $\mathcal{P} \wedge \neg(\forall \alpha \in \mathbb{C}\ (\mathcal{C}_1 \Rightarrow \mathcal{C}_2))$ is $\{1\}$ by calling function `Prove`.

$$\begin{aligned}&\texttt{Prove}[\text{“}\texttt{Knot Heptagon}\text{”}, \texttt{Mapping} \rightarrow \texttt{map}, \texttt{GroebnerBasis} \rightarrow\\ &\quad\{\texttt{CoefficientDomain} \rightarrow \texttt{RationalFunctions},\\ &\quad\texttt{MonomialOrder} \rightarrow \texttt{DegreeReverseLexicographic}\}]\end{aligned} \tag{25}$$

The above call of function `Prove` tells Eos to compute Gröbner basis of the polynomial set $[\![\mathcal{P} \wedge \neg(\forall \alpha \in \mathbb{C}\ (\mathcal{C}_1 \Rightarrow \mathcal{C}_2))]\!]$. Let $\mathcal{V}$ be the set of variables in $[\![\mathcal{P} \wedge \neg(\forall \alpha \in \mathbb{C}\ (\mathcal{C}_1 \Rightarrow \mathcal{C}_2))]\!]$. The Gröbner basis computation is carried out in the domain of polynomials whose variables are in $\mathcal{V} \setminus \{\texttt{wd}\}$ and whose coefficients are in functions of $\mathbb{Q}(\texttt{wd})$. Eos computes Gröbner basis and generates a proof document (whose title is given by the first argument of the call of `Prove`) [16].

6.3 Proof Results

The proofs of (17)–(22) are successful. The CPU time used for Gröbner basis computation on Mac OS X (Intel Core i7 8G 2.4GHz) machine varies from 69.236984 seconds (for proving (17)) to 1994.889588 seconds (for proving (20)).

7 Conclusion

We presented the construction of a regular heptagon using the knot fold. Our method consists in applying 2-fold operation coupled with Huzita's fold operations. The fold lines that make the knot fold are determined by solving geometrical constraints. We further showed the proof of the correctness of the construction of a regular heptagon based on Gröbner bases.

Note that using Huzita's fold operations (O1) $\sim$ (O7), a regular heptagon is constructible. With Eos we need 9 Huzita's fold operations and more than 10 auxiliary operations such as unfolding and marking supporting points. Further investigation of the geometrical constraints by the knot fold is required to construct regular 11-gon, which is impossible by Huzita's fold operations.

Another issue that should be investigated in the future research is the rigidity of the knot fold. We need to define the notion of rigidity in mathematical terms in this knot fold as well as in more general origami that may involve 3D constructions.

7.1 Acknowledgments

We thank Stephen Watt of Western Ontario University for drawing our attention to the knot fold. This work is supported by JSPS KAKENHI Grant No. 25330007. The first author of this paper is supported by a postdoctoral fellowship at Kwansei Gakuin University.

References

[1] D. S. Kasir. *The Algebra of Omar Khayyam (English translation of Khayyam's original work Al-Jabr W'al Muqâbalah).* Teachers College Press, 1931.

[2] P. L. Wantzel. Recherches sur les moyens de connaître si un problème de géométrie peut se résoudre avec la règle et le compas. *Journal de Mathématiques Pures et Appliquées*, pages 366–372, 1837.

[3] H. Huzita. Axiomatic Development of Origami Geometry. In *Proceedings of the First International Meeting of Origami Science and Technology*, pages 143–158, 1989.

[4] R. C. Alperin. A Mathematical Theory of Origami Constructions and Numbers. *New York Journal of Mathematics*, 6:119–133, 2000.

[5] T. Ida, A. Kasem, F. Ghourabi, and H. Takahashi. Morley's Theorem Revisited: Origami Construction and Automated Proof. *Journal of Symbolic Computation*, 46(5):571 – 583, 2011.

[6] J. Justin. Résolution par le pliage de l'équation du troisième degré et applications géométriques. In *Proceedings of the First International Meeting of Origami Science and Technology*, pages 251–261, 1989.

[7] F. Ghourabi, A. Kasem, and C. Kaliszyk. Algebraic Analysis of Huzita's Origami Operations and their Extensions. In *Automated Deduction in Geometry*, LNAI/LNCS. Springer, 2013. (to appear).

[8] F. Ghourabi, T. Ida, H. Takahashi, M. Marin, and A. Kasem. Logical and Algebraic View of Huzita's Origami Axioms with Applications to Computational Origami. In *Proceedings of the 22nd ACM Symposium on Applied Computing (SAC'07)*, pages 767–772, Seoul, Korea, 2007.

[9] R. C. Alperin and R. J. Lang. One-, Two, and Multi-fold Origami Axioms. In *Origami^4, Proceedings of the Fourth International Meeting of Origami Science, Mathematics, and Education (4OSME)*, pages 371–393, 2009.

[10] F. Ghourabi, T. Ida, and H. Takahashi. Computational Origami of Angle Quintisection. In *Proceedings of the Workshop in Symbolic Computation in Software Science (SCSS2008), RISC Technical Report Series*, 08-08, pages 57–68, 2008.

[11] H. M. Cundy and A. P. Rollett. *Mathematical Models.* Oxford University Press, 1961.

[12] J. K. Brunton. Polygonal Knots. *The Mathematical Gazette*, 45(354):299–301, 1961.

[13] K. Sakaguchi. On Polygons Made by Knotting Slips of Paper. *Technical Report of Research Institute of Education, Nara University of Education*, 18:55–58, 1982.

[14] J. Maekawa. Introduction of the study of knot tape. In *Origami^5, Proceedings of the Fourth International Meeting of Origami Science, Mathematics, and Education (5OSME)*, pages 395–403, 2011.

[15] D. A. Cox. *Galois Theory.* Wiley-Interscience, 2004.

[16] F. Ghourabi, T. Ida, and A. Kasem. Proof Documents for Automated Origami Theorem Proving. In *Automated Deduction in Geometry*, volume 6877 of *LNCS*, pages 78–97. Springer, 2011.

Automated Verification of Equivalence on Quantum Cryptographic Protocols

Takahiro Kubota[1], Yoshihiko Kakutani[1], Go Kato[2], Yasuhito Kawano[2] and Hideki Sakurada[2]

[1] The University of Tokyo, Tokyo, Japan
{takahiro.k11_30,kakutani}@is.s.u-tokyo.ac.jp
[2] NTT Communication Science Laboratories, Kanagawa, Japan
{kato.go,kawano.yasuhito,sakurada.hideki}@lab.ntt.co.jp

Abstract

It is recognized that security verification of cryptographic protocols tends to be difficult and in fact, some flaws on protocol designs or security proofs were found after they had been presented. The use of formal methods is a way to deal with such complexity. Especially, process calculi are suitable to describe parallel systems. Bisimilarity, which denotes that two processes behave indistinguishably from the outside, is a key notion in process calculi. However, by-hand verification of bisimilarity is often tedious when the processes have many long branches in their transitions. We developed a software tool to automatically verify bisimulation relation in a quantum process calculus qCCS and applied it to Shor and Preskill's security proof of BB84. The tool succeeds to verify the equivalence of BB84 and an EDP-based protocol, which are discussed in their proof.

1 Introduction

Security proofs of cryptographic protocols tend to be complex and difficult to verify. This fact has been recognized in the classical cryptography and there is a line of researches to deal with the complexity of verification [1, 2]. Since by-hand proofs are prone to human error and time consumption, efforts have been put into automating security proofs and verification in the classical cryptography. Security proofs are also complex in the quantum cryptography, where we must also consider attacks using entanglements. The first security proof of BB84 quantum key distribution (QKD) protocol by Mayers [3] is about 50 pages long. After that paper, researchers have been seeking simpler proofs [4, 5]. Since QKD is one of the closest application to practice in the quantum information field, it is important to examine QKD systems' security formally and make it machine-checkable.

There are several formal frameworks for quantum protocols. Feng et al. developed a process calculus qCCS [6]. In this calculus, a protocol is formalized as a configuration. A configuration is a pair of a process and a quantum state corresponding to quantum variables. Weak open bisimulation relation on qCCS configurations is defined. Weakly open bisimilar configurations have the same transitions up to internal ones and reveal the identical quantum states to the outsider (adversary) in each step. The relation is proven to be closed by parallel composition of processes. A use case of a process calculus for formal verification is to prove weak bisimilarity of implementation and specification described as configurations. qCCS has been applied to quantum teleportation, superdense coding and BB84 protocols.

To extend application of formal methods to security proofs, we applied qCCS to Shor and Preskill's security proof of BB84 [4] in the previous paper [7]. In Shor and Preskill's security proof, security of BB84 is proven to be equivalent to that of another protocol based on an entanglement distillation protocol (EDP), and then the latter is proven to be secure. We formalized BB84 and the EDP-based protocol as configurations and proved their bisimilarity by hand. However, by-hand verification of bisimilarity is often tedious when configurations have many long branches in their transitions. In this paper, we present a software tool to verify bisimilarity of given two qCCS configurations. There are two main contributions of our work.

First, we implemented a software tool that formally verifies bisimilarity of qCCS configurations without recursive structures. A protocol possibly takes security parameters for various purposes. As for BB84, it takes two parameters: one determines the length of qubits which Alice sends to Bob in the first quantum communication, and the other is for tolerated error-rate in the eavesdropping detection step. Such parameters are passed to the verifier as symbols and are interpreted appropriately. In our verifier's framework, quantum states and operations are described as symbols. Since attacks from the adversary are also treated as symbols, our verifier can deal with unspecified attacks. To examine the behavioural equivalence, adversary's views denoted by partial traces must be checked to be equal in each step of a protocol. The verifier automatically checks the equality using user-defined equations.

Second, we applied our verifier to Shor and Preskill's security proof of BB84 [4]. The verifier takes as input two configurations denoting BB84 and the EDP-based protocol, and equations about the properties of error-correcting codes and measurement of the halves of EPR pairs. The verifier succeeds to verify the bisimilarity of the configurations of the two protocols.

The package of the verifier is available from `http://hagi.is.s.u-tokyo.ac.jp/~tk/qccs verifier.tar.gz`. It includes a user manual and and example scripts in the folders `doc` and `scripts`.

Related Work The authors of qCCS presented the notion of symbolic bisimulation for quantum processes [8]. A purpose is to verify bisimilarity algorithmically. They proved the strong (internal actions should be simulated) symbolic bisimilarity is equivalent to the strong open bisimilarity, and actually presented an algorithm to verify symbolic ground (outsiders do not perform quantum operations adaptively) bisimilarity. Since our purpose is to apply a process calculus to security proofs where adversarial interference must be taken into account, we implemented a tool that verifies weak open bisimilarity on the basis of the original qCCS [6].

2 The Verifier

Formal Framework The verifier employs a sublanguage of qCCS, where the syntax of recursion and the use of classical data are restricted. The framework is still expressive, because our target protocols do not need recursion and classical probability distributions can be represented as quantum states denoted by diagonal operators. The syntax of the processes in the verifier is as follows.

$$P ::= \ \texttt{discard(}\tilde{q}\texttt{)} \mid \mathbf{c}?q.P \mid \mathbf{c}!q.P \mid \texttt{meas}\ b\ \texttt{then}\ P\ \texttt{saem} \mid op\texttt{[}\tilde{q}\texttt{]}.P \mid P||P \mid P\backslash L$$

Intuitively, $\texttt{discard(}\tilde{q}\texttt{)}$ means termination of a process keeping a sequence of quantum variables $\tilde{q}$ secret. $\mathbf{c}!q.P$ sends a quantum variable q to a channel $\mathbf{c}$ and then executes P. $\mathbf{c}?q.P$ receives quantum data to a variable q from a channel $\mathbf{c}$ and then executes P. $\texttt{meas}\ b\ \texttt{then}\ P\ \texttt{saem}$ measures a quantum bit b and executes P if the outcome is 0 or terminates if the outcome is 1. $op\texttt{[}\tilde{q}\texttt{]}.P$ performs a superoperator op to quantum variables $\tilde{q}$ and then executes P. $P||P'$ executes P and P' in parallel. $P\backslash L$, where L is a set of channels, executes P keeping channels in L private. Let $\mathrm{qv}(P)$ be the set of quantum variables that occur in a process P. $\mathbf{c}!q.P$, $\texttt{meas}\ b\ \texttt{then}\ P\ \texttt{saem}$, $op\texttt{[}\tilde{q}\texttt{]}.P$ and $P||P'$ are defined only if $q \notin \mathrm{qv}(P)$, $b \in \mathrm{qv}(P)$, $\tilde{q} \subset \mathrm{qv}(P)$ and $\mathrm{qv}(P) \cap \mathrm{qv}(P') = \emptyset$ respectively. In our verifier, each quantum variable is defined with its qubit-length that is indicated by a natural number symbol declared beforehand. The adoption of natural number symbols is important to treat security parameters in the verifier. Quantum states are represented as character strings called environments. The syntax of the environments is given as follows.

$$\rho ::= X\texttt{[}\tilde{q}\texttt{]} \mid op\texttt{[}\tilde{q}\texttt{](}\rho\texttt{)} \mid \rho\,\texttt{*}\,\rho \mid \texttt{proj}i\texttt{[}b\texttt{](}\rho\texttt{)} \mid \texttt{Tr[}\tilde{q}\texttt{](}\rho\texttt{)}$$

Let $\mathrm{qv}(\rho)$ be the set of quantum variables that occur in an environment ρ. $X\texttt{[}\tilde{q}\texttt{]}$ means that quantum variables $\tilde{q}$ are in a quantum state X. $op\texttt{[}\tilde{q}\texttt{](}\rho\texttt{)}$ is a quantum state after the application of operation op to quantum variables $\tilde{q}$ in an environment ρ. $\rho\,\texttt{*}\,\rho'$ represents the tensor product of ρ and ρ' but $\rho\,\texttt{*}\,\rho'$ and $\rho'\,\texttt{*}\,\rho$ are identified in the verifier. $\rho\,\texttt{*}\,\rho'$ is defined only if $\mathrm{qv}(\rho) \cap \mathrm{qv}(\rho') = \emptyset$. $\texttt{proj}i\texttt{[}b\texttt{](}\rho\texttt{)}$ means the reduced density operator $|i\rangle\langle i|_b \rho |i\rangle\langle i|_b$ with $i \in \{0, 1\}$. $\texttt{Tr[}\tilde{q}\texttt{](}\rho\texttt{)}$ means the partial trace of ρ by $\tilde{q}$. We call a pair $\langle P, \rho\rangle$ of a process P and an environment ρ a configuration only if $\mathrm{qv}(P) \subset \mathrm{qv}(\rho)$, because quantum states of qubits occurring in P must be defined in ρ. For example, an agent that sends the halves of n EPR pairs to the outside is formalized as a configuration $\langle \texttt{c!q.discard(r)}, \texttt{EPR[q,r]*ANY[s]}\rangle$, where the quantum variables $\texttt{q}, \texttt{r}$ are of the qubit length n and the quantum variable $\texttt{s}$ is defined with an arbitrary qubit-length. $\texttt{EPR[q,r]}$ is interpreted to $(|00\rangle\langle 00| + |00\rangle\langle 11| + |11\rangle\langle 00| + |11\rangle\langle 11|)^{\otimes n}_{\texttt{q,r}}$ and $\texttt{ANY[s]}$ is arbitrarily for adversary's density operators.

A labelled transition system based on the original one [6] is implemented in the verifier. Transitions are interpreted as interactions between configurations and the adversary. For example, the configuration above performs a transition $\langle \texttt{c!q.discard(r)}, \texttt{EPR[q,r]*ANY[s]}\rangle \xrightarrow{\texttt{c!q}} \langle \texttt{discard(r)}, \texttt{EPR[q,r]*ANY[s]}\rangle$, where the adversary can recognize that the quantum variable $\texttt{q}$ is sent through the channel $\texttt{c}$. Transitions caused by internal actions such as quantum operations, internal communications and conditional branches are labelled τ. The label τ intuitively means that the action is invisible from the outside. The transition rules of conditional branch are $\langle \texttt{meas}\ b\ \texttt{then}\ P\ \texttt{saem}, \rho\rangle \xrightarrow{\tau} \langle P, \texttt{proj0[}b\texttt{](}\rho\texttt{)}\rangle$ and $\langle \texttt{meas}\ b\ \texttt{then}\ P\ \texttt{saem}, \rho\rangle \xrightarrow{\tau} \langle \texttt{discard(}\mathrm{qv}(P)\texttt{)}, \texttt{proj1[}b\texttt{](}\rho\texttt{)}\rangle$. These may decrease the trace of the density operator denoting a quantum state. Since the trace indicates the probability to reach to a configuration, we can restore the probability from configurations to conditional branches. We identify the branches that are evoked by an identical quantum variable. Eventually, probability is absorbed in quantum states and excluded from the transition system.

The transition system of the original qCCS is probabilistic, which is caused by the measurement construction $M[q; x]$ in the syntax [6]. It measures an observable M of a quantum variable q and stores the result in a classical variable x. There are two ways to formalize quantum measurement since one can be also formalized as a superoperator. Because only $M[q; x]$ evokes a probabilistic branch, processes with different formalization of measurement are not bisimilar in general. We proposed a criteria how to select the way in our previous work [7]; we should use $M[q; x]$ if a process behaves differently according to the measurement. For example, the case where a process performs different labeled transitions is typical. Otherwise, we should formalize measurement by a superoperator. In our verifier's framework, $M[q; x]$ cannot be written independently. Instead, the syntax $\texttt{meas}\ b\ \texttt{then}\ P\ \texttt{saem}$ is the composition of $M[q; x]$ and the conditional branch in the original qCCS. This restriction and the conditions on quantum variables prevent the situation where two configurations with the different ways of formalization of measurement are not bisimilar.

Procedure to Check Bisimulation Weak bisimulation relation [6] is defined on qCCS configurations. Roughly speaking, weakly bisimilar configurations (1) perform identical labelled transitions up to τ transitions and (2) reveal identical density operators whatever operations the adversary performs to her quantum variables. The adversary's view is denoted by the partial trace `Tr[`$\tilde{q}$`](`ρ`)` when the global quantum state is ρ and $\tilde{q}$ are the quantum variables that are not revealed to the adversary. The verifier takes as input two configurations and user-defined equations on environments and returns *true* or *false*. It is designed to be sound with respect to the original qCCS, that is, two configurations are bisimilar if the verifier returns *true* with them and some valid equations. This is because

$$\hat{\mathcal{R}}_{eqs} = \{(\mathcal{C}, \mathcal{D}) \mid \text{The verifier returns } \textit{true}\text{, given } \mathcal{C}, \mathcal{D}, \text{and } \textit{eqs}.\}$$

is a bisimulation relation for all valid equations *eqs*. Precisely, $\hat{\mathcal{R}}_{eqs}$ is converted accordingly because `meas` b `then` P `saem` is decomposed to $M[b;x]$ and a conditional branch, which performs a two-step transition in the branch with $x = 0$. The recursive procedure to verify bisimilarity is as follows. Since the transitions of the processes written in our sublanguage are finite, the procedure always terminates.

1. The procedure takes as input two configurations $\langle P_0, \rho_0\rangle$, $\langle Q_0, \sigma_0\rangle$ and user-defined equations on environments.

2. If P_0 and Q_0 can perform any τ-transitions of superoperator applications, they are all performed at this point. Let $\langle P, \rho\rangle$ and $\langle Q, \sigma\rangle$ be the obtained configurations.

3. Whether $\mathrm{qv}(P) = \mathrm{qv}(Q)$ is checked. If it does not hold, the procedure returns *false*.

4. Whether `Tr[`$\mathrm{qv}(P)$`](`ρ`)` $=$ `Tr[`$\mathrm{qv}(Q)$`](`σ`)` is checked with user-defined equations. The procedure to check the equality of quantum states are described in the next subsection. If it does not hold, the procedure returns *false*.

5. A new superoperator symbol $\mathcal{E}$`[`$\mathrm{qv}(\rho) - \mathrm{qv}(P)$`]` that stands for an adversarial operation is generated.

6. For each $\langle P', \rho'\rangle$ such that $\langle P, \mathcal{E}\texttt{[}\mathrm{qv}(\rho) - \mathrm{qv}(P)\texttt{](}\rho\texttt{)}\rangle \xrightarrow{\alpha} \langle P', \rho'\rangle$, the procedure checks whether there exists $\langle Q', \sigma'\rangle$ such that $\langle Q, \mathcal{E}\texttt{[}\mathrm{qv}(\sigma) - \mathrm{qv}(Q)\texttt{](}\sigma\texttt{)}\rangle \xrightarrow{\tau*}\xrightarrow{\alpha}\xrightarrow{\tau*} \langle Q', \sigma'\rangle$ and the procedure returns *true* with the input $\langle P', \rho'\rangle$ and $\langle Q', \sigma'\rangle$. If there exists, it goes to the next step 7. Otherwise, it returns *false*.

7. For each $\langle Q', \sigma'\rangle$ such that $\langle Q, \mathcal{E}\texttt{[}\mathrm{qv}(\sigma) - \mathrm{qv}(Q)\texttt{](}\sigma\texttt{)}\rangle \xrightarrow{\alpha} \langle Q', \sigma'\rangle$, the procedure checks whether there exists $\langle P', \rho'\rangle$ such that $\langle P, \mathcal{E}\texttt{[}\mathrm{qv}(\rho) - \mathrm{qv}(P)\texttt{](}\rho\texttt{)}\rangle \xrightarrow{\tau*}\xrightarrow{\alpha}\xrightarrow{\tau*} \langle P', \rho'\rangle$ and the procedure returns *true* with the input $\langle Q', \sigma'\rangle$ and $\langle P', \rho'\rangle$. If there exists, it returns *true*. Otherwise, it returns *false*.

The step 2 prominently reduces the search space. Indeed, $\langle op_1[\tilde{q}].P||op_2[\tilde{r}].Q, \rho\rangle$ and $\langle P||Q, \mathcal{F}^{\tilde{r}}_{op_2}(\mathcal{E}^{\tilde{q}}_{op_1}(\rho))\rangle$ are bisimilar, and $\mathcal{F}^{\tilde{r}}_{op_2}(\mathcal{E}^{\tilde{q}}_{op_1}(\rho)) = \mathcal{E}^{\tilde{q}}_{op_1}(\mathcal{F}^{\tilde{r}}_{op_2}(\rho))$ holds because $\tilde{q} \cap \tilde{r} = \emptyset$ holds. Therefore, superoperators that can originally cause τ transitions are disposed.

Equality Test of Quantum States To test the equality of given two environments, the verifier checks whether they are transformed to the same form. There are two kinds of transformations, trace out and application of user-defined rules.

Partial trace is significant to test the equality of quantum states. For example, two different quantum states `op1[q](X[q]*Y[r]*Z[s])` and `op2[r](U[q]*V[r]*Z[s])` have the same partial trace `Z[s]` under `Tr[q,r]` for arbitrary interpretation of the superoperators `op1`, `op2` and the quantum states `X`, `Y`, `Z`, `U`, `V`. The verifier eliminates symbols of superoperators and quantum states according to the partial trace rules below.

$\texttt{Tr[}\tilde{q}\texttt{](}\mathcal{E}\texttt{[}\tilde{r}\texttt{](}\rho\texttt{))} = \texttt{Tr[}\tilde{q}\texttt{](}\rho\texttt{)}$ if $\tilde{r} \subset \tilde{q}$ (1), $\texttt{Tr[}\tilde{q}\texttt{](}\rho\texttt{)} = \texttt{Tr[}\tilde{r}\texttt{](Tr[}\tilde{s}\texttt{](}\rho\texttt{))}$ if $\tilde{q} = \tilde{r} \cup \tilde{s}$ (2),
$\texttt{Tr[}\tilde{q}\texttt{](}\mathcal{E}\texttt{[}\tilde{r}\texttt{](}\rho\texttt{))} = \mathcal{E}\texttt{[}\tilde{r}\texttt{](Tr[}\tilde{q}\texttt{](}\rho\texttt{))}$ if $\tilde{q} \cap \tilde{r} = \emptyset$ (3), $\texttt{Tr[}\tilde{q}\texttt{](}\rho * \rho_{\tilde{q}} * \sigma\texttt{)} = \rho * \sigma$ (4).

If $\tilde{r} \subset \tilde{q}$ holds, $\texttt{Tr[}\tilde{q}\texttt{](}\mathcal{E}\texttt{[}\tilde{r}\texttt{](}\rho\texttt{))}$ is rewritten to $\texttt{Tr[}\tilde{q}\texttt{](}\rho\texttt{)}$ eliminating $\mathcal{E}\texttt{[}\tilde{r}\texttt{]}$ by the rule (1); otherwise, by the rules (2) and (3), the trace-out symbol with quantum variables that are disjoint to targets of superoperator goes inside of it's application. After eliminating superoperators, quantum states are traced out by the rule (4).

If an objective quantum state has a part that matches to the left-hand side of a user-defined equation, the part is rewritten to the right-hand side. To apply a user-defined equation, the verifier automatically solves commutativity of superoperators or partial traces that are applied to disjoint sets of quantum variables. For example, if the objective quantum state is `Tr[q](hadamard[s] (EPR[q,r]*X[s]))` and a user defines an equation `Tr[q](EPR[q,r])=Tr[q](PROB [q,r])` (E1), the application goes as follows.

$$\begin{aligned}\texttt{Tr[q](hadamard[s](EPR[q,r]*X[s]))} &= \texttt{hadamards[s](Tr[q](EPR[q,r]*X[s]))} && \text{(by 5)}\\ &= \texttt{hadamard[s](Tr[q](PROB[q,r]*X[s]))} && \text{(by E1)}\end{aligned}$$

Since the trace-out rules may have become applicable after applications of user-defined rules, the trace-out procedure is applied again. In each opportunity to test the equality of quantum states, each user-defined equation is applied only once. This guarantees whatever rules a user defines, the equality test terminates.

3 Application to Shor and Preskill's Security Proof

The two protocols BB84 and the EDP-based protocol [4] are formalized as qCCS configurations on the basis of our previous work [7]. Readers can find the script `scripts/shor-preskill.scr` in the package. The interpretations of the symbols of the quantum states and the superoperators that are used in the formalization are also included as `doc/shor-preskill_symbols.pdf`.

On Channels As in general QKD protocols, three kinds of channels are used: public quantum channels, private classical channels and public no-interpolate classical channels. Since the syntax has channel restriction $P\backslash L$, formalization of private channels is straightforward. Public no-interpolate classical channels are realized by copying the data. If a quantum variable q where a classical value is assigned is sent through a public no-interpolate channel c, this is formalized as $...\texttt{copy[}q\texttt{,}Q\texttt{]}.c\texttt{!}q.d\texttt{!}Q...\backslash\{...,c,...\}$, where Q is a new quantum variable, a superoperator `copy` copies the value of q to Q and d is a new non-restricted channel. q will securely sent through the restricted channel c and an adversary obtains the same value accessing Q through a public channel d. Note that the value of q can be copied because it is classical.

Equations for the Proof We defined 10 equations in the verifier. Equations are interpreted to those on density operators and proven correct. For example, the equation below tells the verifier that the halves of EPR pairs are indistinguishable from the uniform distribution. This is used to check that the adversary's views are identical in the EDP-based protocol and BB84 before Alice in the EDP-based protocol measures her halves of the check qubits.

```
equation E7
  Tr[q1_A](EPR[q1_A,q2_A]) = Tr[q1_A](PROB[q1_A,q2_A])
end
```

The user-defined rule above is interpreted to an equation $\mathrm{tr}_{\{\texttt{q1\_A}\}}(|00\rangle\langle 00| + |00\rangle\langle 11| + |11\rangle\langle 00| + |11\rangle\langle 11|)^{\otimes n}_{\texttt{q1\_A,q2\_A}} = \mathrm{tr}_{\{\texttt{q1\_A}\}}(|00\rangle\langle 00| + |11\rangle\langle 11|)^{\otimes n}_{\texttt{q1\_A,q2\_A}}$. The other rules are related to CSS codes, partial traces and measurement of quantum states. We obtained them by formalizing the equality of quantum states that are discussed in the original proof [4].

It is actually natural to express quantum states as symbols with some equations in inputs. In cryptographic protocols, the dimension of quantum states and superoperators cannot be fixed

as a certain number as they depend on security parameters. In addition, qubits going through public channels are potentially interpolated by the adversary. Since arbitrary computation by the adversary must be considered, it should be described as an unspecified superoperator. Therefore, quantum states are difficult to be expressed as concrete matrices, which can be numerically analyzed. Although quantum states are treated symbolically, automatic calculation of partial traces is still possible focusing on occurrence of quantum variables in environments.
Experiment Result We ran the verifier with the input of `shor-preskill.scr`. We used a note PC with Intel Core i5 CPU M 460 @ 2.53GHz and 1GB memory. The transition tree of the EDP-based protocol has 621 nodes and 165 paths, and that of BB84 has 588 nodes and 165 paths. The verifier checked the bisimilarity of the two protocols in 15.98 seconds. The recursive procedure was called 951 times.

4 Conclusions

We presented a software tool to check bisimulation of qCCS configurations and applied it to a security proof of BB84 [4]. In security proofs, equivalence on protocols is often discussed. It can be described as bisimulation relation but it is nearly impossible to check by hand if state transitions of processes have many long branches. In addition, the equality of an adversary's view between two protocols has to be checked in each step. An adversary's view is calculated from a global quantum states, which is possibly denoted by a huge matrix. The verifier does exhausting parts of proofs of behavioral equivalence, namely, it checks the correspondence of all state transitions up to invisible ones and an adversary's view up to equations. On the other hand, a user only have to examine the correctness of formalization of protocols and validity of equations that he defines. It could be difficult to find all appropriate equations for a proof immediately. The verifier is also able to show environments and/or processes when the check procedure returns *false*. With the information, a user can modify equations to input.
Future Work We plan to define probabilistic bisimilarity, which denotes that configurations behave indistinguishably up to some probability. This will be helpful to prove "indistinguishability up to negligible probability" of protocols. It is quite common argument in the cryptography.

References

[1] S. Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, 2005.

[2] B. Blanchet, A. D. Jaggard, A. Scedrov, and J.-K. Tsay. Computationally sound mechanized proofs for basic and public-key kerberos. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, pages 87–99, 2008.

[3] D. Mayers. Unconditional security in quantum cryptography. *J. ACM*, 48:351–406, May 2001.

[4] P. W. Shor and J. Preskill. Simple proof of security of the bb84 quantum key distribution protocol. *Phys. Rev. Lett.*, 85(2):441–444, Jul 2000.

[5] H.-K. Lo and H. F. Chau. Unconditional security of quantum key distribution over arbitrarily long distances. *Phys. Rev. Lett.*, 283(5410):2050–2056, Mar 1999.

[6] Y. Deng and Y. Feng. Open bisimulation for quantum processes. In *Theoretical Computer Science*, volume 7604 of *LNCS*, pages 119–133. 2012.

[7] T. Kubota, Y. Kakutani, G. Kato, Y. Kawano, and H. Sakurada. Application of a process calculus to security proofs of quantum protocols. In *Proceedings of WORLDCOMP/FCS2012*, Jul 2012.

[8] Y. Feng, Y. Deng, and M. Ying. Symbolic bisimulation for quantum processes. arXiv preprint arXiv:1202.3484, 2012.

A Modified Parallel F4 Algorithm for Shared and Distributed Memory Architectures

Severin Neumann

Fakultät für Mathematik und Informatik
Universität Passau, D-94030 Passau, Germany
neumans@fim.uni-passau.de

Abstract

In applications of symbolic computation an often required but complex procedure is finding Gröbner bases for polynomial ideals. Hence it is obvious to realize parallel algorithms to compute them. There are already flavours of the F4 algorithm like [4] and [13] using the special structure of the occurring matrices to speed up the reductions on parallel architectures with shared memory. In this paper we start from these and present modifications allowing efficient computations of Gröbner bases on systems with distributed memory. To achieve this we concentrate on the following objectives: decreasing the memory consumption and avoiding communication overhead. We remove not required steps of the reduction, split the columns of the matrix in blocks for distribution and review the effectiveness of the `SIMPLIFY` function. Finally we evaluate benchmarks with up to 256 distributed threads of an implementation being available at `https://github.com/svrnm/parallelGBC`.

1 Introduction

Parallelization is one of the most used methods to improve the performance of existing software. But it should be introduced systematically. One chooses the most time-consuming segments of a program and tries to improve these first. In applications of symbolic computation an often required but complex procedure is finding Gröbner bases for polynomial ideals. Therefore it is obvious to realize parallel versions of algorithms to compute them. Although it is possible to use Buchberger's algorithm [14], we favour Faugère's algorithm F4 [3] since the reduction is done due to matrix transformation which is known for being well parallelizable.

There are already parallel flavours of F4 like [4] and [13] using the special structure of the occurring matrices to speed up the reduction. Both approaches have been developed for multicore and multiprocessor systems with shared memory. For these systems the number of parallel processing units is limited and currently there are not many platforms serving more than 64 processors. Compared with that clusters of several suchlike computers can have an theoretical unlimited number of processors. This advantage comes with the downside of distributed memory: realizing algorithms requires to consider that data has to be transfered. By that memory duplication and communication overhead are introduced.

In the following we start from the mentioned developments for shared memory systems and present modifications which will finally allow efficient computation of Gröbner bases also for systems with distributed memory. To achieve this we have concentrated on the following objectives: reducing the memory consumption and avoiding communication overhead. We remove needless steps of the reduction, split the columns of the matrix in blocks for distribution and review the effectiveness of the `SIMPLIFY` function.

At the end we will evaluate benchmarks of an implementation using eight distributed nodes having 32 processor cores each. The source code is available at `https://github.com/svrnm/parallelGBC`.

2 Notations

In the following we will use the notation of [11]. Hence $\mathbb{K}$ denotes a field and $\mathbb{K}[x_1, \ldots, x_n]$ the polynomial ring over $\mathbb{K}$ in n indeterminantes. $\mathbb{T}^n$ is defined as the set of all terms $x_1^{\alpha_1} \cdot \ldots \cdot x_n^{\alpha_n}$. σ is a term ordering on this set. For a polynomial $f \in \mathbb{K}[x_1, \ldots, x_n]$ with $f = \sum_{i=1}^{m} c_i \cdot t_i$, where $c_i \in \mathbb{K}$ and $t_i \in \mathbb{T}^n$ for all i, we call the set $Supp(f) := \{t_1, \ldots, t_m\}$ support of f. Then the leading term of f is defined as $LT_\sigma(f) := t_k = max_\sigma(Supp(f))$ and the leading coefficient is $LC_\sigma(f) := c_k$.

Let $\mathcal{F} := \{f_1, \ldots, f_s\} \in \mathbb{K}[x_1, \ldots, x_n] \setminus \{0\}$ be a set of polynomials. Then $I := \langle \mathcal{F} \rangle$ is the ideal generated by $\mathcal{F}$. The elements of the set $\mathbb{B} := \{(i,j) \mid 1 \leq i < j < s\}$ are called critical pairs. For each critical pair exists an S-polynomial $S_{i,j} := \frac{1}{LC_\sigma(f_i)} \cdot t_{i,j} \cdot f_i - \frac{1}{LC_\sigma(f_j)} \cdot t_{j,i} \cdot f_j$ with $t_{i,j} := \frac{lcm(LT_\sigma(f_i), LT_\sigma(f_j))}{LT_\sigma(f_i)}$ and $t_{j,i} := \frac{lcm(LT_\sigma(f_j), LT_\sigma(f_i))}{LT_\sigma(f_j)}$.

The set $\mathcal{F}$ is a σ-Gröbner basis of I if for all critical pairs the S-polynomials can be reduced to zero by the elements of $\mathcal{F}$ with respect to σ.

If not stated otherwise we will use the degree reverse lexicographic termordering (`DegRevLex`) and the finite field with 32003 elements ($\mathbb{F}_{32003}$) for examples and benchmarks.

3 Preliminaries

The F4 algorithm was introduced by Faugère in 1999 [3]. By using the special structure of the matrix and column-based parallelism Faugère and Lachartre introduced a parallelization improving the performance remarkable [4]. We have presented another modification [13] for F4 speeding up the computation by row-based parallelization.

In the following we give a quick summary about Gröbner bases computation and the mentioned developments. For a given set of polynomials $\mathcal{F} := \{f_1, \ldots, f_s\} \in \mathbb{K}[x_1, \ldots, x_n] \setminus \{0\}$ generating an ideal $I = \langle \mathcal{F} \rangle$ a σ-Gröbner basis $\mathcal{G} = \{g_1, \ldots, g_t\} \in \mathbb{K}[x_1, \ldots, x_n]$ can be computed using the F4 algorithm by the following four steps:

1. Create and update the set of critical pairs $\mathbb{B}$ using Gebauer's and Möller's `UPDATE` function [6]. This function guarantees that the leading terms $t_{i,j} \cdot f_i = t_{j,i} \cdot f_j$ of the minuend and subtrahend of the S-polynomial is unique among the critical pairs.

2. Select a subset of all critical pairs for reduction using the normal or the sugar cube strategy [7]. At this point the details are not relevant. For our benchmarks we chose the sugar cube strategy. Although we have to mention that our implementation supports the normal strategy and for some of the polynomial systems we used as example this strategy might perform better.

3. Symbolic preprocess the selected pairs to obtain a coefficient matrix representing the S-polynomials. The matrix is additionally filled with rows representing reduction polynomials for each term which can be reduced using elements of the partial computed Gröbner basis $\mathcal{G}$. The `SIMPLIFY` method can be applied to reuse results of previous reductions.

4. Reduce the generated matrix until row-echelon form. This is equivalent to top-reducing the S-polynomials. All non-zero rows of this matrix having leading terms which are not divisible by any element of $\mathcal{G}$ are inserted into $\mathcal{G}$. As long as new elements are found the algorithm continues with step 1.

The most time-consuming step is the reduction of the coefficient matrix. The mentioned parallelizations of Lachartre and Faugère as well as our own approach concentrate on speeding up this step. We want to show how to use these improvements also for distributed parallel architectures, therefore we have to review them.

Starting after symbolic preprocessing we obtain a coefficient matrix M which has to be reduced to a row-echelon form. M has n rows and m columns with $m \geq n$ and can be decomposed in submatrices:

1. Submatrices S_1 and S_2 arise from the subtrahends and minuends of the S-polynomials. Since the `UPDATE` function guarantees that all polynomials have distinct leading terms the representing rows have unique pivot columns. Therefore S_1 and S_2 are upper-triangular.
2. Submatrix R represents the reduction polynomials. They were computed during symbolic preprocessing. For each reducible term there is only one reduction polynomial. Because of that R is upper-triangular.
3. Finally these submatrices can be split up in pivot and non-pivot columns. The pivots of R and S_1 are forming the submatrix A and the non-pivots the submatrix B. The submatrix S_2 is split into C containing its pivots and D containing the non-pivot columns. This notation was introduced in [4].

The following example demonstrates the construction of a matrix occurring during Gröbner bases computations.

Example 1. *Let $K := \mathbb{K}_{32003}$, $P := K[x_1, x_2, x_3]$ and let the polynomials $g_1 = x_1^2 + x_2^2$, $g_2 = x_1 \cdot x_2 + x_2^2 + x_2 \cdot x_3$ and $g_3 = x_2^2 + x_3^2 + x_3$ form the ideal $I := \langle g_1, g_2, g_3 \rangle$ for which a Gröbner basis with respect to `DegRevLex` should be computed. Using `UPDATE` we get the critical pairs $(2,3)$ and $(1,2)$. They have the same sugar degree $sugar(g_2, g_3) = 3 = sugar(g_1, g_2)$. So we reduce the following polynomials:*

$$\begin{aligned} f_{1,2} &= (x_1^2 + x_2^2) \cdot x_2 = x_1^2 \cdot x_2 + x_2^3 \\ f_{2,1} &= (x_1 \cdot x_2 + x_2^2 + x_2 \cdot x_3) \cdot x_1 = x_1^2 \cdot x_2 + x_1 \cdot x_2^2 + x_1 \cdot x_2 \cdot x_3 \\ f_{2,3} &= (x_1 \cdot x_2 + x_2^2 + x_2 \cdot x_3) \cdot x_2 = x_1 \cdot x_2^2 + x_2^3 + x_2^2 \cdot x_3 \\ f_{3,2} &= (x_2^2 + x_3^2 + x_3) \cdot x_1 = x_1 \cdot x_2^2 + x_1 \cdot x_3^2 + x_1 \cdot x_3 \end{aligned}$$

and as set of reduction polynomials we obtain:

$$\begin{aligned} r_1 &= g_3 \cdot x_2 = x_2^3 + x_2 \cdot x_3^2 + x_2 \cdot x_3 \\ r_2 &= g_2 \cdot x_3 = x_1 \cdot x_2 \cdot x_3 + x_2^2 \cdot x_3 + x_2 \cdot x_3^2 \\ r_3 &= g_3 \cdot x_3 = x_2^2 \cdot x_3 + x_3^3 + x_3^2 \end{aligned}$$

At the end we get the following matrix M:

$$\begin{array}{c} f_{1,2} \\ f_{2,3} \\ r_1 \\ r_2 \\ r_3 \\ f_{2,1} \\ f_{3,2} \end{array} \left(\begin{array}{ccccc|cccccc} \mathbf{1} & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \mathbf{1} & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 1 & 0 & 0 & 1 \\ \hline 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{array}\right)$$

One can easily see the mentioned structure of the matrix. The upper left submatrix is A and its pivots are highlighted. Below is C and the non-pivot columns are forming B and D.

Using this decomposition the (full) reduction of the matrix can be computed using the following algorithm:

1. Primary reduction: Reduce B to $A^{-1} \cdot B$.
2. Secondary reduction: Reduce D to $D - C \cdot A^{-1} \cdot B$.
3. Final reduction: Reduce D to its reduced row-echelon form using Gaussian elimination.
4. Optional reduction: Reduce B using the pivots of D.

Especially the primary reduction can be efficiently parallelized since the matrix is in upper triangular form and so all pivots are known. The primary and secondary reduction can be computed in parallel because a reduced row of B can be applied independently on rows of D.

After the final reduction all non-zero rows of D are elements of the Gröbner basis and can be used to compute further critical pairs. Matrix (AB) does not contain any required elements because by construction the pivots of A are equivalent to leading terms which are already contained in the Gröbner basis. Hence the fourth step is optional and only required if `SIMPLIFY` is used.

There are two different approaches to achieve parallelization: Column-based as suggested by Faugère and Lachatre and row-based as suggested by us. Actually these two are not mutually exclusive and we will use this property to advance the parallelism on architectures with distributed memory. In the following we will use the row-based approach for shared memory parallelism but one can easily see that all modifications are also applicable to the column-based approach of Faugère and Lachatre.

4 Matrix Distribution

In the case of the matrix reduction during Gröbner basis computation we have to consider how to distribute the coefficient matrices with least duplication and communication. Like any other parallel and distributed algorithm for transforming matrices into row echelon form we can choose between row- and column-based distribution or a combination of both.

If row-based or combined distribution is chosen the algorithm scatters rows or blocks among the computation nodes and these need to send rows from one to each other if they are required for further reductions. To reduce the communication overhead this needs to use a strategy which decides how the matrix is distributed. Since we use rows-based parallelization for the shared memory level and since the column-based distribution does play well with the special properties of Gröbner basis computations we didn't analyse the effectiveness of the row-based or combined approach.

If column-based distribution is chosen the pivot matrices A and C have to be send to all nodes and the non-pivot matrices B and D can be distributed without duplication. Since all pivots are preset for primary and secondary reduction this approach does only require communication before and after all reductions are done.

Furthermore the matrices are sparse and by using appropriate matrix formats the communication overhead can be reduced even more. We chose to store the off-diagonal elements of A and C in a coordinate list, i.e. there is one list storing tuples of row, column and value. This format allows reordering in parallel executable sets to improve the row-based parallelization as presented in [13]. For B and D a modified list of lists format is used storing pairs of row index and value for each column. This allows to distribute columns independently. As suggested in [8, Chapter 6] the columns are distributed round-robin to optimize load balancing. Afterwards each node can use row-based parallelism on the local shared memory. The following shows the distribution of the matrix constructed in example 1:

Example 2. *Using two parallel nodes the matrix M of the previous example is decomposed into one coordinate list and two lists of lists. Afterwards the pivots are sent to both nodes and the two sets of non pivot columns are distributed among the nodes. Finally the two nodes are holding the following data:*

Node #1

(A C)		
(4,1,1)	(4,3,1)	(2,0,1)
(2,1,1)	(3,5,1)	
(0,5,1)		
(1,5,1)		
(1,6,1)		

(B D)	
5	(6,1)
7	(4,1)
9	(4,3)

Node #2

(A C)		
(4,1,1)	(4,3,1)	(2,0,1)
(2,1,1)	(3,5,1)	
(0,5,1)		
(1,5,1)		
(1,6,1)		

(B D)		
6	(2,1)	(3,1)
8	(6,1)	-
10	(4,1)	-

Table 1 shows that the final reduction can be computed on a single node because the submatrix D consumes only a tiny fraction of the whole matrix. One can easily check that this is true in general: the submatrices S_1 and S_2 have s rows each and the submatrix R has r rows. There are p pivot columns and n non-pivot columns. Hence the coefficient matrix has $2 \cdot s + r$ rows and $p + n$ columns. After primary and secondary reduction the final reduction only requires the submatrix D which has s remaining rows and n columns. Therefore all reduced columns can be gathered on one node which executes the Gaussian elimination on D.

Polynomial System	Sugar degree	primary matrix	final matrix	ratio (%)
Gametwo 7 [10]	17	7731 x 5818	2671x2182	13
Cyclic 8 [1]	14	3819 x 4244	612x1283	4.8
Cyclic 9	16	70251 x 75040	4014x10605	0.8
Katsura 12 [9]	9	12141 x 11490	2064x2155	3.2
Katsura 13	9	26063 x 23531	4962x4346	3.5
Eco 12 [12]	9	12547 x 10937	2394x1269	2.2
Eco 13	10	25707 x 24686	2883x2310	1.0
F966 [5]	9	60274 x 63503	4823x9058	1.1

Table 1: Ratio of primary matrix and final matrix during reduction for selected problems.

The optional reduction is an obstacle for the effectiveness of the algorithm since it requires that the reduced matrix D has to be distributed again to be applied on B. Because of that we examine if the optional reduction is useful at all. Even more we put to test, if the `SIMPLIFY` method and distributed computation play well together.

5 Simplifying the F4 Algorithm

Removing the optional reduction may scale down the required communication overhead and the reduction time. Even more, if `SIMPLIFY` isn't used during symbolic preprocessing, the distributed matrix has not to be gathered completely since the columns of the reduced matrix B are not required after primary and secondary reduction.

Algorithm 1: `SIMPLIFY`

```
    Input: $t \in \mathbb{T}^n$ a term, $f \in \mathbb{K}[x_1, \ldots, x_n]$ a polynomial, $\mathcal{F} \in (F_k)_{k=1,\ldots,d-1)}$, where $F_k$ is
           finite subset of $\mathbb{K}[x_1, \ldots, x_n]$
    Output: a non evaluated product, i.e. an element of $\mathbb{T}^n \times \mathbb{K}[x_1, \ldots, x_n]$
1   foreach $u \in$ list of divisors of $t$ do
2       if $\exists j (1 \leq j < d)$ such that $(u \cdot f) \in F_j$ then
3           $\tilde{F}_j$ is the row echelon form of $F_j$ w.r.t. $\sigma$ ;
4           there exists a (unique) $p \in \tilde{F}_j^{+}$ such that $\mathrm{LT}_\sigma(p) = \mathrm{LT}_\sigma(u \cdot f)$ ;
5           if $u \neq t$ then return SIMPLIFY($\frac{t}{u}, p, \mathcal{F}$) else return $(1, p)$;
6       end
7   end
```

The `SIMPLIFY` method does not guarantee that a reduction is improved by reusing previous results. Actually `SIMPLIFY` proves it effectiveness mostly by application. Hence we require a best possible implementation of the method. Algorithm 1 is a repetition of the original `SIMPLIFY` [3]. It leaves some details open and there are some possibilities for optimization:

- Searching through all divisors of t is time consuming regarding the fact that only some divisors of u will satisfy the condition $(u, f) \in F_j$ for $j \in (1 \leq j < d)$.
- To check if there exists a j with (u, f) in F_j it is important how F_j is stored. A naive implementation will require to loop over all rows of F_j until the condition is satisfied.
- This algorithm does not check if there is another $j' \neq j$ satisfying the condition and provides a better replacement.

To address these issues we propose the following modifications:

- All rows of $\tilde{F}_1 \ldots, \tilde{F}_{d-1}$ are stored in a two-dimensional map. The first key is a polynomial f and the second is a term u. This allows to update (u, f) if a *better* replacement is found. By that only one value per pair is stored. This will decrease the required memory.
- The first map should provide a fast random access to the value of key f and the second map should have an ordering to allow a decreasing iteration of the possible terms. In this way the search space for possible replacements is reduced to the multiples of f only.
- The value of a map element is the reduced form of $f \cdot u$.

We call this data structure `SimplifyDB`. We can provide an improved version of `SIMPLIFY` as shown by algorithm 2. Additionally we have to introduce a function to update the `SimplifyDB`. Therefore after reduction we execute algorithm 3 for each row of $\tilde{F}_j^{+}$. The insertion step allows us to compare candidates for the `SimplifyDB`. In our implementation for two candidates the *better* is the one which has less elements in its support. This is equivalent to the first weighted length function $wlen(p) = \#Supp(p)$ in [2]. At this point we didn't try any other suggested weighted length function.

`SIMPLIFY` can be improved even more. During symbolic preprocessing it is called twice: once for the selected critical pair (m, f) with $m \in \mathbb{T}^n$ and $f \in \mathcal{G}$ and once to create reduction

Algorithm 2: `SIMPLIFY` with `SimplifyDB`

```
Input: $t \in \mathbb{T}^n$ a term, $f \in \mathbb{K}[x_1, \ldots, x_n]$ a polynomial, SimplifyDB
Output: a non evaluated product, i.e. an element of $\mathbb{T}^n \times \mathbb{K}[x_1, \ldots, x_n]$
1 foreach $u \leq t \in$ SimplifyDB[$f$] do
2     if $u$ divides $t$ then
3         $p =$ SimplifyDB[$f$][$u$] ;
4         if $u \neq t$ then return SIMPLIFY($\frac{t}{u}$, $p$, SimplifyDB) else return $(1,p)$ ;
5     end
6 end
```

Algorithm 3: Insertion into `SimplifyDB`

```
Input: $t \in \mathbb{T}^n$ a term, $f, p \in \mathbb{K}[x_1, \ldots, x_n]$ polynomials, SimplifyDB
Output: SimplifyDB
1 if $\exists u \in$ SimplifyDB[f] : $t = u$ then
2     Replace SimplifyDB[$f$][$u$] with $p$ if it is better ;
3 else
4     SimplifyDB[$f$][$t$] $= p$ ;
5 end
```

polynomials $m' \cdot f$ with $m' \in \mathbb{T}^n$ and $f' \in \mathcal{G}$. As one can easily see the second input parameter is always an element of $\mathcal{G}$. So it stands to reason to restrict `SIMPLIFY`:

$$\texttt{SIMPLIFY}(t, i), \text{ where } i \text{ is the index of } g_i \in \mathcal{G}$$

Still there is the recursive call of `SIMPLIFY` taking a polynomial p as parameter which might not be an element of $\mathcal{G}$. Recall that p was found in the `SimplifyDB` using f and a term $u = \frac{t}{z}$, with $z \in \mathbb{T}^n$. With the knowledge, that $f = g_i \in \mathcal{G}$ we can write $p = f \cdot u - r = g_i \cdot u - r$, where $r \in \mathbb{K}[x_1, \ldots, x_n]$ is the sum of all reductions applied on f during a previous reduction step. The recursive call is looking for another polynomial p' which simplifies $z \cdot p = z \cdot (g_i \cdot u - r)$. If such a polynomial p' exists we can write $p' = z \cdot (g_i \cdot u - r) - r' = z \cdot u \cdot g_i - (z \cdot r - r')$. Hence if the `SimplifyDB` stores p' at index $(g_i, z \cdot u)$ instead of (p, z) the recursion is obsolete and `SIMPLIFY` can be replaced with algorithm 4 being recursion free.

Algorithm 4: New `SIMPLIFY` algorithm

```
Input: $t \in \mathbb{T}^n$ a term, $i$ is the index of $g_i \in \mathcal{G}$, SimplifyDB
Output: a non evaluated product, i.e. an element of $\mathbb{T}^n \times \mathbb{K}[x_1, \ldots, x_n]$
1 foreach $u \leq t \in$ SimplifyDB[f] do
2     if $u$ divides $t$ then return ($\frac{t}{u}$, SimplifyDB[$f$][$u$]) ;
3 end
```

Before benchmarking different variants of the distributed versions of the algorithm a first look at timings using only shared memory will thin out the number of combinations. For the following computations we used a system with 48 AMD Opteron™ 6172 cores having 64 gigabyte of main memory. The implementation of the parallelization for the shared memory is presented in [13] and in its latest version it is realized using Intel® Threading Building Blocks (TBB) and Open Multi-Processing (OpenMP) for parallelization.

Table 2 shows that the optional reduction is mostly useless and can be left out during Gröbner basis computation. Table 3 shows that our new recursion-free `SIMPLIFY` is also in practice faster and even more memory efficient.

Finally `SIMPLIFY` is improving performance by increasing memory usage and so for solving larger polynomial systems it might be possible to find a solution within the bounds of available memory by disabling `SIMPLIFY`. Consequently we will compare computations with and without the improved `SIMPLIFY` for the distributed parallel implementation in the following section.

Polynomial systems	without optional reduction		with optional reduction	
	max. matrix size	runtime	max. matrix size	runtime
Gametwo 7	7808 x 5895	28.46	7731 x 5818	29.85
Cyclic 8	3841 x 4295	11.18	3819 x 4244	12.15
Cyclic 9	70292 x 75080	606.7	70251 x 75040	628.3
Katsura 12	12142 x 11491	34.49	12141 x 11490	36.66
Katsura 13	26063 x 23531	152.2	26063 x 23531	169.2
Eco 12	12575 x 11349	25.22	12547 x 10937	28.36
Eco 13	25707 x 24686	91.76	25707 x 24686	103.7
F966	60898 x 63942	98.32	60274 x 63503	118.5

Table 2: Comparison of computations with and without optional reduction. The new `SIMPLIFY` algorithm was used in both cases.

Polynomial	with new `SIMPLIFY`		with original `SIMPLIFY`		without `SIMPLIFY`	
systems	runtime (s)	memory	runtime	memory	runtime	memory
Gametwo 7	**28.46**	459 MB	28.82	721 MB	29.97	262 MB
Cyclic 8	**11.18**	262 MB	14.43	393 MB	12.28	131 MB
Cyclic 9	606.7	15.6 GB	649.2	24.0 GB	**542.9**	2.10 GB
Katsura 12	**34.49**	655 MB	39.28	1.57 GB	43.74	328 MB
Katsura 13	**152.2**	2.56 GB	187.2	7.27 GB	207.1	917 MB
Eco 12	**25.22**	328 MB	38.75	1.38 GB	50.87	262 MB
Eco 13	**91.76**	1.14 GB	136.4	5.77 GB	218.2	852 MB
F966	**98.32**	2.56 GB	218.2	11.3 GB	212.5	1.25 GB

Table 3: Comparison of original and new `SIMPLIFY`

6 Benchmarks

For the following benchmarks of our distributed implementation we used a cluster of up to eight systems having 16 Intel® Xeon® E5-2670 cores with hyperthreading and 64 GB of RAM allowing us using up to 256 parallel threads. For data distribution we use the Message Passing Interface (MPI) and the Boost.MPI wrapper.

Table 4 compares once again runtimes for computations with and without `SIMPLIFY` using one, two or four nodes. For all polynomial input systems the computation time goes up and the distribution has no positive effect. However we accomplished to speed-up the reduction time. This is not an inconsistency. The overall runtime goes up due to the communication overhead and the possibility to improve the performance is limited to decreasing the reduction time. Hence the problem is the *small* size of the given polynomial systems. Figure 1 illustrates this issue for Cyclic 9.

Table 5 adds further and more difficult examples. Especially for Katsura 15 and Cyclic 10 the distributed parallelization takes effect providing a speed-up of 2.3 and 2.7 for the overall computation time and 5.7 and 5.0 for the reduction using four respectively eight nodes. Excluding the Eco 14 system the computation is most effective without SIMPLIFY and in particular for Cyclic 10 computations were not even possible with it due to memory overflows.

Eco 14 is an counterexample because it is not possible to decrease the computation time using a distributed system. This is caused by the effectiveness of SIMPLIFY for the problem instance.

The crashed computations for Katsura 15 and for Cyclic 10 without SIMPLIFY with two nodes are caused by a not yet solved overflow in the MPI implementation if more than 512 MB have to be transfered in one operation. This does not happen with four or eight nodes because the matrix size for each node decreases by factor two respectively four.

At the end no distributed computation with SIMPLIFY is faster and hence it should be removed in further developments of distributed parallel algorithms for Gröbner bases computations.

Polynomial systems	with SIMPLIFY			without SIMPLIFY		
# nodes	1	2	4	1	2	4
Gametwo 7	29.85	34.55	56.03	**28.48**	38.53	28.88
	(5.549)	(6.709)	(6.010)	(6.630)	(4.607)	(**3.427**)
Cyclic 8	7.798	11.12	9.520	**7.497**	14.06	20.78
	(1.327)	(1.707)	(1.953)	(1.332)	(1.473)	(**1.168**)
Cyclic 9	506.0	518.2	494.7	**330.6**	361.7	427.6
	(222.2)	(195.2)	(138.8)	(149.9)	(122.4)	(**73.73**)
Katsura 12	42.00	46.36	55.23	**28.59**	39.97	66.85
	(6.083)	(8.220)	(7.762)	(6.699)	(4.535)	(**3.362**)
Katsura 13	187.4	190.0	186.4	**139.6**	158.8	176.0
	(40.46)	(46.61)	(36.82)	(50.03)	(31.07)	(**18.15**)
Eco 12	**21.35**	29.40	41.49	24.98	41.08	88.02
	(**2.564**)	(3.760)	(3.953)	(6.458)	(3.664)	(3.097)
Eco 13	**91.78**	106.5	120.9	111.4	157.5	212.6
	(**12.10**)	(16.32)	(14.62)	(38.65)	(23.10)	(14.67)
F966	**91.88**	106.2	132.1	93.60	139.2	185.1
	(20.86)	(26.29)	(23.91)	(38.59)	(26.32)	(**18.70**)

Table 4: Distributed computation (and reduction) times for 1,2 or 4 nodes

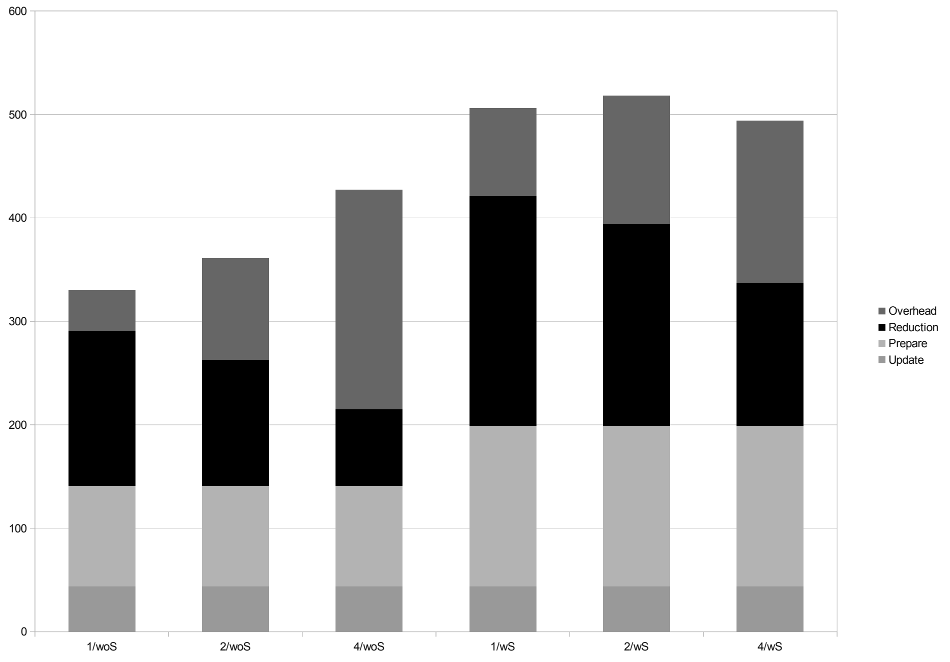


Figure 1: Computation time in detail for Cyclic 9 with and without `SIMPLIFY`

Poly. sys.	with `SIMPLIFY`				without `SIMPLIFY`			
# nodes	1	2	4	8	1	2	4	8
Katsura 14	957.5	831.0	748.9	744.8	775.6	**659.9**	675.4	991
	(313.9)	(263.4)	(201.2)	169.1	(398.4)	(224.4)	(135.1)	(**86**)
Katsura 15	5719	*crashed*	3440	3252	5615	3857	**3252**	3372
	(2571)	(-)	1272	1008	(3781)	(1960)	(1094)	(**666**)
Cyclic 10	*crashed*	*crashed*	*crashed*	*crashed*	30400	*crashed*	13270	**11280**
	(-)	(-)	(-)	(-)	(25240)	(-)	(7862)	(**5010**)
Eco 14	**509.8**	519.4	552.6	626	800.5	787.0	996.7	1660
	(110.6)	(117.7)	(98.02)	(89.38)	(411.3)	(224.0)	(127.7)	(**78.24**)

Table 5: Distributed computation (and reduction) times for larger input systems.

7 Conclusion

We have shown that the parallelization of the F4 algorithm can be expanded form a shared memory system to a cluster of distributed nodes providing a multiple of the computation power of a single system. This was primarily achieved by using column-based parallelism, removing the optional reduction and the `SIMPLIFY` function. It allowed us to solve larger input systems like Katsura 15 or Cyclic 10 almost three times faster using four or respectively eight distributed nodes. Some work is still required to optimize the speed-up and the communication of the nodes has to be improved to make solving of even larger problem instances possible.

References

[1] Göran Björck and Uffe Haagerup. All cyclic p-roots of index 3, found by symmetry-preserving calculations, 2008.

[2] Michael Brickenstein. Slimgb: Gröbner bases with slim polynomials. *Revista Matemática Complutense*, 23(2):453–466, 2010.

[3] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases (F4). *Journal of Pure and Applied Algebra*, 139(1–3):61–88, June 1999.

[4] Jean-Charles Faugère and Sylvain Lachartre. Parallel Gaussian Elimination for Gröbner bases computations in finite fields. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, pages 89–97, New York, USA, July 2010. ACM.

[5] Jean-Charles Faugère, Moreau Francois De Saint Martin, and Fabrice Rouillier. Design of regular nonseparable bidimensional wavelets using Groebner basis techniques. *IEEE Transactions on Signal Processing*, 46(4):845–856, 1998.

[6] Rüdiger Gebauer and H. Michael Möller. On an installation of buchberger's algorithm. *Journal of Symbolic Computation*, 6:275–286, December 1988.

[7] Alessandro Giovini, Teo Mora, Gianfranco Niesi, Lorenzo Robbiano, and Carlo Traverso. "One sugar cube, please" or selection strategies in the buchberger algorithm. In *Proceedings of the 1991 international symposium on Symbolic and algebraic computation*, ISSAC '91, pages 49–54, New York, USA, 1991. ACM.

[8] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 1996.

[9] Shigetoshi Katsura, Wataru Fukuda, Sakari Inawashiro, Nahomi Fujiki, and Rüdiger Gebauer. Distribution of effective field in the ising spin glass of the $\pm j$ model at $t = 0$. *Cell Biochemistry and Biophysics*, 11:309–319, 1987.

[10] David M. Kreps and Robert Wilson. Sequential equilibria. *Econometrica*, 50(4):863–94, July 1982.

[11] Martin Kreuzer and Lorenzo Robbiano. *Computational Commutative Algebra 1*. Computational Commutative Algebra. Springer, 2000.

[12] Alexander Morgan. *Solving polynomial systems using continuation for engineering and scientific problems*. Classics in applied mathematics. SIAM, 2009.

[13] Severin Neumann. Parallel reduction of matrices in Gröbner bases computations. In Vladimir P. Gerdt, Wolfram Koepf, Ernst W. Mayr, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing*, volume 7442 of *Lecture Notes in Computer Science*, pages 260–270. Springer Berlin Heidelberg, 2012.

[14] Kurt Siegl. A parallel factorization tree gröbner basis algorithm. In *Parallel Symbolic Computation PASCO, 1994: Proceedings of the First International Symposium*, River Edge, USA, 1994. World Scientific Publishing Co., Inc.

Generating Asymptotically Non-Terminant Initial Variable Values for Linear Diagonalizable Programs

Rachid Rebiha[1]*, Nadir Matringe[2,3] and Arnaldo Vieira Moura[1]

[1] Institute of Computing UNICAMP, University of Campinas, SP. Brasil.
`rachid@ic.unicamp.br`
[2] Universit de Poitiers, Laboratoire Mathmatiques et Applications, France.
[3] Institue de Mathematiques de Jussieu, Université Paris 7-Denis Diderot, France.

Abstract

We present the key notion of *asymptotically non-terminant initial variable values* for non-terminant loop programs. We show that those specific values are directly associated to inital variable values for which the corresponding loop program does not terminate. Considering linear diagonalizable programs, we describe powerful computational methods that generate automatically and symbolically a semi-linear space represented by a linear system of equalities and inequalities. Each element of this space provides us with asymptotically non-terminant initial variable values. Our approach is based on linear algebraic methods. We obtain specific conditions using certain basis and matrix encodings related to the loop conditions and instructions.

1 Introduction

Research on formal methods for program verification [12, 15, 8, 17] aims at discovering mathematical techniques and developing their associated algorithms to establish the correctness of software, hardware, concurrent systems, embedded systems or hybrid systems. Static program analysis [12, 15], is used to check that a software is free of defects, such as buffer overflows or segmentation faults, which are safety properties, or termination and non-termination, which are liveness properties. Proving termination of while loop programs is necessary for the verification of liveness properties that any well behaved and engineered system, or any safety critical embedded system must guarantee. We could list here many verification approaches that are only practical depending on the facility with which termination can be automatically determined. Verification of temporal properties of infinite state systems [20] is another example.

Recent work on automated termination analysis of imperative loop programs has focused on a partial decision procedure based on the discovery and synthesis of ranking functions. Such functions map the loop variable to a well-defined domain where their value decreases further at each iteration of the loop [9, 10]. Several interesting approaches, based on the generation of *linear* ranking functions, have been proposed [3, 4] for loop programs where the guards and the instructions can be expressed in a logic supporting linear arithmetic. For the generation of such functions, there are effective heuristics [14, 10] and, in some cases, there are also complete methods for the synthesis of linear ranking functions [16]. On the other hand, it is easy to generate a simple linear terminant loop program that does not have a linear ranking function. And in this case such complete synthesis methods [16] fail to provide a conclusion about the termination or the non termination of such a program.

*Supported by FAPESP grant 2011/08947-1 and FAPESP grant BEPE 2013/04734-9.

In this work we address the non-termination problem for linear while loop programs. In other words, we consider the class of loop programs where the loop condition is a conjunction of linear inequalities and the assignments to each of the variables in the loop instruction block, are affine or linear forms. In matrix notation, *linear loop programs* will be represented in a general form as: while $(Bx > b)$, $\{x := Ax + c\}$ (i.e., A and B are matrices, b and c are vectors of real numbers, and that x is a vector of variables.). Without loss of generality, the termination/non-termination analysis for such a class of linear programs could be reduced to the problem of termination/non-termination for homogeneous linear programs [6, 21]. Those being programs where linear assignments consist of homogeneous expressions, and where the linear loop condition consists of at most one inequality. Concerning effective program transformations and simplification techniques, non-termination analysis for programs presented in a more complex form can often be reduced to an analysis of a program expressed in this basic affine form. Despite tremendous progress over the years [6, 5, 7, 13, 11, 2, 1], the problem of finding a practical, sound and complete methods for determining or analyzing non termination remains very challenging for this class of programs, and for all initial variable values. We started our investigation from our preliminary technical reports [19, 18] where we introduced a termination analysis in which algorithms ran in polynomial time complexity. Here, considering a non terminating loop, we introduce new static analysis methods that compute automatically, and in polynomial time complexity, the set of initial input variable values for which the program does not terminate, and also a set of initial inputs values for which the program does terminate. This justifies the innovation of our contributions, *i.e.*, none of the other mentioned related work is capable of generating such critical information over non-terminating loops. We summarize our contributions as follows:

- To the best of our knowledge, we introduce a new key notion for non-termination and termination analysis for loop programs: we identify the important concept of *asymptotically non-terminant initial variable values*, ANT for short. Any asymptotically non-terminant initial variable values can be directly related to an initial variable value for which the considered program does not terminate.
- Our theoretical contributions provide us with efficient computational methods running in polynomial time complexity and allowing the exact computation of the set of all asymptotically non-terminant initial variable values for a given loop program.
- We generate automatically a set of linear equalities and inequalities describing a semi-linear space that represents symbolically the set of all asymptotically non-terminant initial variable values. The set of ANT values contains the set of non-terminant initial variable values. On the other hand the complementary set of ANT values is a precise under-approximation of the set of terminant inputs for the same program.

Example 1.1. (Motivating Example) *Consider the following program depicted below on the left. We show a part of the output of our algorithm below on the right.*

(i) Pseudo code:

```
while(2x+3y-z>0){
   x:= y + z;
   y:=-(1/2)x+(3/2)y-(1/2)z;
   z:=(3/2)x-(3/2)y+(1/2)z;}
```

(ii) Output of our prototype:

```
Locus of ANT
[[4u[1]+u[2]+u[3]>0]
AND[u[1]+4u[2]+4u[3]>0]
AND[-u[1]+u[2]-u[3]=0]]
OR[[-u[1]+u[2]-u[3]>0]]
```

The semi-linear $ANT = \{u = (u_1, u_2, u_3)^\top \in E \mid 4u_1 + u_2 + u_3 > 0 \wedge u_1 + 4u_2 + 4u_3 > 0 \wedge -u_1 + u_2 - u_3 = 0\} \cup \{u = (u_1, u_2, u_3)^\top \in E \mid -u_1 + u_2 - u_3 > 0\}$ *represents symbolically all asymptotically initial variable values that are directly associated to initial variable values for which the program does not terminate. On the other hand, the complementary of this set* co-$ANT = \{u = (u_1, u_2, u_3)^\top \in E \mid 4u_1 + u_2 + u_3 \leq 0 \vee u_1 + 4u_2 + 4u_3 \leq 0 \vee -u_1 + u_2 - u_3 \neq 0\} \cap \{u = (u_1, u_2, u_3)^\top \in E \mid -u_1 + u_2 - u_3 \leq 0\}$ *is a precise under-approximation of the set of all initial variable values on which the program terminates.* □

The rest of this article is organized as follows. Section 2 can be seen as a preliminary section where we introduce some key notions and results from linear algebra, which will be used to build our computational methods. In this section, we also present our computational model for programs and some further notations. Section 3 introduces the new notion of *asymptotically non-terminant initial variable values*. Section 4 provides the main theoretical contributions of this work. This section also presents our computational methods that generate a symbolic representation of the asymptotically non-terminant variable values for linear programs. We provide a complete dicussion in Section 5. Finally, Section 6 states our conclusions.

2 Linear Algebra and Linear Loop Programs

Here, we present key linear algebraic notions and results which are central in the theoretical and algorithmic development of our methods. We denote by $\mathcal{M}(m, n, \mathbb{K})$ the set of $m \times n$ matrices with entries in $\mathbb{K}$, and simply $\mathcal{M}(n, \mathbb{K})$ when $m = n$. The Kernel of A, also called its *nullspace*, denoted by $Ker(A)$, is $Ker(A) = \{v \in \mathbb{K}^n \mid A \cdot v = 0_{\mathbb{K}^m}\}$. In fact, when we deal with square matrices, these Kernels are *Eigenspaces*. Let A be a $n \times n$ square matrix with entries in $\mathbb{K}$. A nonzero vector $x \in \mathbb{K}$ is an eigenvector for A associated with the eigenvalue $\lambda \in \mathbb{K}$ if $A \cdot x = \lambda x$, i.e., $(A - \lambda I_n) \cdot x = 0$ where I_n is the $n \times n$ identity matrix. The nullspace of $(A - \lambda I_n)$ is called the *eigenspace* of A associated with eigenvalue λ. Let n be a positive integer, we will denote $\mathbb{R}^n$ by E and its canonical basis by $B_c = (e_1, \ldots, e_n)$. Let A be a square matrix in $\mathcal{M}(n, \mathbb{R})$. Let us introduce the notation $Spec(A)$ for the set of eigenvalues of A in $\mathbb{R}$, and we will write $Spec(A)^*$ for the set $Spec(A) - \{0\}$. For $\lambda \in Spec(A)$, we will denote by $E_\lambda(A)$ the corresponding eigenspace. Throughout the paper we write d_λ for the dimension of $E_\lambda(A)$. We will say that A is diagonalizable if $E = \oplus_{\lambda \in Spec(A)} E_\lambda(A)$. Let A be a diagonalizable matrix. For each eigenspace $E_\lambda(A)$, we will take a basis $B_\lambda = (e_{\lambda,1}, \ldots, e_{\lambda,d_\lambda})$, and we define

$$B = \cup_{\lambda \in Spec(A)} B_\lambda$$

as a basis for E.

Definition 2.1. *Let x belong to E. We denote by x_λ its component in $E_\lambda(A)$. If x admits the decomposition $\sum_{\lambda \in Spec(A)} (\sum_{i=1}^{d_\lambda} x_{\lambda,i} e_{\lambda,i})$ in B, we have $x_\lambda = \sum_{i=1}^{d_\lambda} x_{\lambda,i} e_{\lambda,i}$.* □

We denote by P the transformation matrix corresponding to B, whose columns are the vectors of B, decomposed in B_c. Letting d_i denote the integer d_{λ_i} for the ease of notation, we recall the following lemma.

Lemma 2.1. *We have $P^{-1}AP = diag(\lambda_1 Id_1, \ldots, \lambda_t Id_t)$. We denote by D the matrix $P^{-1}AP$.* □

As we will obtain our results using the decomposition of x in B, we recall how one obtains it from the decomposition of x in B_c.

Lemma 2.2. *Let $x \in E$. If $x = \sum_{i=1}^{n} x_i e_i = (x_1, ..., x_n)^\top \in B_c$, and x decomposes as $\sum_{j=1}^{t}(\sum_{i=1}^{d_j} x_{\lambda_j,i} e_{\lambda_j,i})$ in B, the coefficients $x_{\lambda_j,i}$ are those of the column vector $P^{-1}x$ in B_c.* □

Throughout the paper we write $< , >$ for the canonical scalar product on E, which is given by $< x, y >= \sum_{i=1}^{n} x_i y_i$, and recall, as it is standard in static program analysis, that a primed symbol x' refers to the next state value of x after a transition is taken. Next, we present *transition systems* as representations of imperative programs and *automata* as their computational models.

Definition 2.2. *A* transition system *is given by $\langle x, L, \mathcal{T}, l_0, \Theta \rangle$, where $x = (x_1, ..., x_n)$ is a set of variables, L is a set of locations and $l_0 \in L$ is the initial location. A* state *is given by an interpretation of the variables in x. A* transition *$\tau \in \mathcal{T}$ is given by a tuple $\langle l_{pre}, l_{post}, q_\tau, \rho_\tau \rangle$, where l_{pre} and l_{post} designate the pre- and post- locations of τ, respectively, and the transition relation ρ_τ is a first-order assertion over $x \cup x'$. The transition guard q_τ is a conjunction of inequalities over x. Θ is the initial condition, given as a first-order assertion over x. The transition system is said to be* linear *when ρ_τ is an affine form.* □

We will use the following matrix notations to represent loop programs and their associated transitions systems.

Definition 2.3. *Let P be a loop program represented by the transition system $\langle x = (x_1, ..., x_n), l_0, \mathcal{T} = \langle l, l, q_\tau, \rho_\tau \rangle, l_0, \Theta \rangle$. We say that P is a* linear loop program *if the following conditions hold:*

- *Transition guards are conjunctions of linear inequalities. We represent the loop condition in matrix form as $Vx > b$ where $V \in \mathcal{M}(m, n, \mathbb{R})$ and $b \in \mathbb{R}^m$. By $Vx > b$ we mean that each coordinate of vector Vx is greater than the corresponding coordinate of vector b.*
- *Transition relations are affine or linear forms. We represent the linear assignments in matrix form as $x := Ax + c$, where $A \in \mathcal{M}(n, \mathbb{R})$ and $c \in \mathbb{R}^n$.*

The linear loop program *$P = P(A, V, b, c)$ will be represented in its most general form as* while $(Vx > b)$, $\{x := Ax + c\}$. □

In this work, we use the following linear loop program classifications.

Definition 2.4. *We identify the following three types of linear loop programs, from the more specific to the more general form:*

- Homogeneous*: We denote by $P^{\mathbb{H}}$ the set of programs where all linear assignments consist of* homogeneous *expressions, and where the linear loop condition consists of at most one inequality. If P is in $P^{\mathbb{H}}$, then P will be written in matrix form as* while $(< v, x >> 0)$, $\{x := Ax\}$*, where v is a $(n \times 1)$-vector corresponding to the loop condition, and where $A \in \mathcal{M}(n, \mathbb{R})$ is related to the list of assignments in the loop. We say that P has a* homogeneous *form and it will also be denoted as $P(A, v)$.*
- Generalized Condition*: We denote by $P^{\mathbb{G}}$ the set of linear loop programs where the loop condition is* generalized *to a conjunction of multiple linear homogeneous inequalities.*
- Affine Form*: We denote by P^{A} the set of loop programs where the inequalities and the assignments are generalized to affine expressions. If P is in P^{A}, it will be written as* while $(Vx > b)$, $\{x := Ax + c\}$*, for A and V in $\mathcal{M}(n, \mathbb{R})$, $b \in \mathbb{R}^m$, and $c \in \mathbb{R}^n$.*

□

Without lost of generality, the static analysis for the class of linear programs P^{A} could be reduced to the problem of termination/non-termination for homogeneous linear programs in $P^{\mathbb{H}}$. In this work we consider programs in $P^{\mathbb{H}}$. The generalization to programs in P^{A} can already be done and it will be reported elsewhere.

3 Asymptotically Non-terminant Variable Values

In this section we present the new notion of *asymptotically non-terminant* variable values. It will prove to be central in analysis of termination, in general. Let A be a matrix in $\mathcal{M}(n, \mathbb{R})$, and v be a vector in E. Consider the program $P(A, v)$: while $(< v, x >> 0)$, $\{x := Ax\}$, which takes values in E. We first give the definition of the termination of such a program.

Definition 3.1. *The program $P(A, v)$ is said to be terminating on $x \in E$, if and only if $< v, A^k(x) >$ is not positive, for some $k \geq 0$.* □

In other words, Definition 3.1 states that if the program $P(A, v)$ performs $k \geq 0$ loop iterations from initial variables $x$in E, we obtain $x := A^k(x)$. Thus, if $< v, A^k(x) > \leq 0$, the loop condition is violated, and so $P(a, v)$ terminates. Next, we introduce the following important notion.

Definition 3.2. *We say that $x \in E$ is an asymptotically non terminating value for $P(A, v)$ if there exists $k_x \geq 0$ such that $P(A, v)$ is non terminating on $A^{k_x}(v)$. We will write that x is ANT for $P(A, v)$, or simply x is ANT. We will also write that $P(A, v)$ is ANT on x.* □

Note that if $P(A, v)$ is non terminating on $A^{k_x}(x)$ then $< v, A^k(x) >$ is > 0 for $k \geq k_x$.

The following result follows directly from the previous definition.

Corollary 3.1. *An element $x \in E$ is ANT if and only if $< v, A^k(x) >$ is positive for k large enough.* □

If the set of ANT points is not empty, we say that the program $P(A, v)$ is ANT. We will also write NT for non terminant. For the programs we study here, the following lemma already shows the importance of such a set.

Lemma 3.1. *The program $P(A, v)$ is NT if and only if it is ANT.* □

Proof. It is clear that NT implies ANT (i.e., $NT \subseteq ANT$), as a NT point of $P(A, v)$ is of course ANT (with $k_x = 0$). Conversely, if $P(A, v)$ is ANT, call x an ANT point, then $A^{k_x}(x)$ is a NT point of $P(A, v)$, and so $P(A, v)$ is NT. □

As one can easily see, the set of NT points is included in the set of ANT points. But the most important property of the ANT set remains in the fact that each of its point provides us with an associated element in NT for the corresponding program. In other words, each element x in the ANT set, even if it does not necessarily belong to the NT set, refers directly to initial variable values $y_x = A^{k_{[x]}}(x)$ for which the program does not terminate, i.e., y_x is an NT point. We can say that there exists a number of loop iterations $k_{[x]}$, departing from the initial variable values x, such that $A^{k_{[x]}}(x)$ correspond to initial variable values for which $P(A, v)$ does not terminate. But, it does not necessarily implies that x is also an NT point for $P(A, v)$. In fact, program $P(A, v)$ could terminate on the initial variable values x by performing a number of loop iterations strictly smaller than $k_{[x]}$. On the other hand, the complementary set co-ANT provides us with a quite precise under approximation of the set of all initial variable values for which the program terminates.

The set of ANT points of a program is also important to the understanding of the termination of a program with more than one conditional linear inequality, as well as for termination over the rationals, for example. This will be reported elsewhere.

4 Automated generation of ANT loci

In this section we show how we generate automatically and exactly the *ANT Locus*, i.e., the set of all *ANT* points for linear diagonalizable programs over the reals. We represent symbolically the computed *ANT Locus* by a semi-linear space defined by linear equalities and inequalities. Consider the program $P(A, v)$: while $(< v, x >> 0)$, $\{x := Ax\}$. The study of *ANT* sets depends on the spectrum of A. Recall the introductory discussion and Definition 2.1, at Section 2.

Proposition 4.1. *For x in E, and $k \geq 1$, the scalar product $< v, A^k(x) >$ is equal to*

$$\sum_{\lambda \in Spec(A)} \lambda^k < v, x_\lambda >= \sum_{\lambda \in Spec(A)^*} \lambda^k < v, x_\lambda > . \quad \square$$

Proof. It is a direct consequence of the equality $A^k(x_\lambda) = \lambda^k x_\lambda$. □

4.1 The regular case

We first analyze the situation where A is what we call regular:

Definition 4.1. *We say that A is regular, if $Spec(A) \cap \big[-Spec(A)\big]$ is an empty set, i.e.: if λ belongs to $Spec(A)$, then $-\lambda$ does not belong to $Spec(A)$.* □

In this case, we make the following observation:

Proposition 4.2. *Let μ be the nonzero eigenvalue of largest absolute value, if it exists, such that $< v, x_\mu >$ is nonzero. For k large, the quantity $< v, A^k(x) >$ is equivalent to $\mu^k < v, x_\mu >$.* □

Proof. Indeed, we can write $< v, A^k(x) >$ as

$$\mu^k < v, x_\mu > + \sum_{\{\lambda, |\lambda| < |\mu|\}} \lambda^k < v, x_\lambda >= \mu^k(< v, x_\mu > + \sum_{\{\lambda, |\lambda| < |\mu|\}} \frac{\lambda^k}{\mu^k} < v, x_\lambda >),$$

and $\frac{\lambda^k}{\mu^k}$ approaches zero when k goes to infinity. □

We then define the following sets.

Definition 4.2. *For μ a* ***positive*** *eigenvalue of A, we define S_μ as follows:*

$$S_\mu = \{x \in E, < v, x_\mu >> 0, < v, x_\lambda >= 0 \ for \ |\lambda| > |\mu|\}$$

□

In order to obtain S_μ for all **positive** eigenvalues μ of A, one needs to calculate $< v, x_\mu >$, and $< v, x_\lambda >$ for all eigenvalues λ such that $|\lambda| > |\mu|$. For all eigenvalues λ involved in the computation of S_μ, one also needs to evaluate the coefficients $c_{\lambda,i} =< v, e_{\lambda,i} >$ for all eigenvectors $e_{\lambda,i} \in B$ and $1 \leq i \leq d_\lambda$. Thus, we have $< v, x_\lambda >= \sum_{i=1}^{d_\lambda} c_{\lambda,i} x_{\lambda,i}$. Now, we only need to compute the coefficient $x_{\lambda,i}$, which are those of the column vector $P^{-1}x$ in B_c where P is the transformation matrix corresponding to B. See Lemma 2.2.

We ca now state the following theorem, allowing the exact computation of the ANT Locus for the regular case.

Theorem 4.1. *The program $P(A, v)$ is ANT on x if and only if x belongs to*

$$\bigcup_{\mu>0\in Spec(A)} S_\mu.$$

□

Proof. Let x belong to E. If all $< v, x_\lambda >$ are zero for $\lambda \in Spec(A)$, then $< v, A^k(x) >= 0$ and x is not ANT. Otherwise, let μ be the eigenvalue of largest absolute value, such that $< v, x_\mu >$ is nonzero. Then, according to Proposition 4.2, the sign of $< v, A^k(x) >$ is the sign of $\mu^k < v, x_\mu >$ for k large. If μ is negative, this sign will be alternatively positive and negative, depending on the parity of k, and x is not ANT. If μ is positive, this sign will be the sign of $< v, x_\mu >$, hence x will be ANT if and only if $< v, x_\mu >$ is positive. This proves the result. □

Example 4.1. (Running example) *Consider the program $P(A, v)$ depicted as follows:*

(i) Pseudo code:

```
while(x+y-2z>0){
  x:= x-4y-4z;
  y:= 8x-11y-8z;
  z:= -8x+8y+5z;}
```

(ii) Associated matrices:

$$A = \begin{pmatrix} 1 & -4 & -4 \\ 8 & -11 & -8 \\ -8 & 8 & 5 \end{pmatrix}, \text{ and } v = \begin{pmatrix} 1 \\ 1 \\ -2 \end{pmatrix}.$$

Step 1*: Diagonalization of the matrix A:*

$$P = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 1 & 0 \\ -2 & 0 & 1 \end{pmatrix}, D = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -3 & 0 \\ 0 & 0 & -3 \end{pmatrix} \text{ and } P^{-1} = \begin{pmatrix} 1 & -1 & -1 \\ -2 & 3 & 2 \\ 2 & -2 & -1 \end{pmatrix}.$$

Using our notations, the obtained eigenvectors (i.e., the column vectors of P) are denoted as follows: $e_{1,1} = (1, 2, -2)^\top$; $e_{-3,1} = (1, 1, 0)^\top$; $e_{-3,2} = (1, 0, 1)^\top$.

Step 2*: We compute S_μ for all positive $\mu \in Spec(A)^*$:*

- *We compute first the coefficients $c_{\lambda,i}$:*
  $c_{1,1} =< v, e_{1,1} >=< (1, 1, -2)^\top, (1, 2, -2)^\top >= 7$,
  $c_{-3,1} =< v, e_{-3,1} >=< (1, 1, -2)^\top, (1, 1, 0)^\top >= 2$,
  $c_{-3,2} =< v, e_{-3,1} >=< (1, 1, -2)^\top, (1, 0, 1)^\top >= -1$

- *We compute the coefficient $x_{\lambda,i}$, which are those of the column vector $P^{-1} \cdot u$ in B_c, where $u = (u_1, u_2, u_3)$ is the vector encoding the initial variable values.*

$$P^{-1}.\begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} u_1 - u_2 - u_3) \\ -2u_1 + 3u_2 + 2u_3) \\ 2u_1 - 2u_2 - u_3) \end{pmatrix} = \begin{pmatrix} x_{1,1} \\ x_{-3,1} \\ x_{-3,2} \end{pmatrix}.$$

We can now proceed to the generation of the linear constraints defining a semi-linear space

describing symbolically and exactly S_μ.
$< v, x_1 >= c_{1,1}x_{1,1} = 7(u_1 - u_2 - u_3)$
$< v, x_{-3} >= c_{-3,1}x_{-3,1} + c_{-3,2}x_{-3,2} = -6u_1 + 8u_2 + 5u_3)$
Hence, we have: $S_1 = \{u = (u_1, u_2, u_3)^\top \in E \mid (u_1 - u_2 - u_3 > 0) \wedge (-6u_1 + 8u_2 + 5u_3 = 0)\}$.

Step 3*: We apply Theorem 4.1 to generate the ANT Locus. It reduces to the semi-linear space* $S_1 = \{u = (u_1, u_2, u_3)^\top \in E \mid (u_1 - u_2 - u_3 > 0) \wedge (-6u_1 + 8u_2 + 5u_3 = 0)\}$. □

4.2 The general case: handling linear diagonalizable programs

For the general case, in the asymptotic expansion of $< v, A^k(x) >$ one can have compensations between $\lambda^k < v, x_\lambda >$ and $(-\lambda)^k < v, x_\lambda >$, as these terms can be zero when k is even, for instance, and of a well determined sign when k is odd. We thus need to take care of this issue. To this end, we introduce the following notation.

Definition 4.3. *If λ does not belong to Spec(A), for any $x \in E$, we write $x_\lambda = 0$.* □

We have the following propositions, which give the asymptotic behavior of $< v, A^k(x) >$.

Proposition 4.3. *Let μ be the nonzero eigenvalue of largest absolute value, if it exists, such that $< v, x_{|\mu|} + x_{-|\mu|} >$ is nonzero. For k large, the quantity $< v, A^{2k}(x) >$ is equivalent to $|\mu|^{2k}(< v, x_{|\mu|} + x_{-|\mu|} >)$.* □

Proof. Indeed, we can write $< v, A^{2k}(x) >$ as

$$\mu^{2k}(< v, x_{|\mu|} > + < v, x_{-|\mu|} >) + \sum_{\{|\lambda|, |\lambda| < |\mu|\}} \lambda^{2k}(< v, x_{|\lambda|} > + < v, x_{-|\lambda|} >)$$

$$= \mu^{2k}(< v, x_{|\mu|} > + < v, x_{-|\mu|} >) + \sum_{\{|\lambda|, |\lambda| < |\mu|\}} \frac{\lambda^{2k}}{\mu^{2k}}(< v, x_{|\lambda|} > + < v, x_{-|\lambda|} >)).$$

and $\frac{\lambda^k}{\mu^k}$ approaches to zero when k goes to infinity. □

Proposition 4.4. *Let μ be the nonzero eigenvalue of largest absolute value, if it exists, such that $< v, x_{|\mu|} - x_{-|\mu|} >$ is nonzero. For k large, the quantity $< v, A^{2k+1}(x) >$ is equivalent to $|\mu|^{2k+1}(< v, x_{|\mu|} - x_{-|\mu|} >)$.* □

Proof. The proof is similar to the proof of Proposition 4.3. □

As in the previous Section, we introduce the following relevant sets.

Definition 4.4. *For $|\mu|$ in $|Spec(A)^*|$, we define the sets $S^0_{|\mu|}$ and $S^1_{|\mu|}$ as follows:*

$$S^0_{|\mu|} = \{x \in E, < v, x_{|\mu|} + x_{-|\mu|} >> 0, < v, x_{|\lambda|} + x_{-|\lambda|} >= 0 \ for \ |\lambda| > |\mu|\}, \ and$$

$$S^1_{|\mu|} = \{x \in E, < v, x_{|\mu|} - x_{-|\mu|} >> 0, < v, x_{|\lambda|} - x_{-|\lambda|} >= 0 \ for \ |\lambda| > |\mu|\}.$$

□

In order to compute the sets $S^0_{|\mu|}$ and $S^1_{|\mu|}$, we obtain the coefficients $c_{\lambda,i} =< v, e_{\lambda,i} >$ for all eigenvalues λ. If the λ appearing as index in the coefficient $c_{\lambda,i}$ is not an eigenvalue, then we fix $c_{\lambda,i} = 0$ and $d_\lambda = 0$ (see Definition 4.3). Thus, we have $< v, x_{|\lambda|} + x_{-|\lambda|} >= \sum_{i=1}^{d_{|\lambda|}} c_{|\lambda|,i}x_{|\lambda|,i} + \sum_{i=1}^{d_{-|\lambda|}} c_{-|\lambda|,i}x_{-|\lambda|,i}$, and $< v, x_{|\lambda|} - x_{-|\lambda|} >= \sum_{i=1}^{d_{|\lambda|}} c_{|\lambda|,i}x_{|\lambda|,i} - \sum_{i=1}^{d_{-|\lambda|}} c_{-|\lambda|,i}x_{-|\lambda|,i}$.

We finally obtain the following main theorem.

Theorem 4.2. *The program $P(A, v)$ is ANT on x if and only if x belongs to the set*

$$\bigcup_{(|\mu|,|\mu'|)\in|Spec(A)^*|\times|Spec(A)^*|} S^0_{|\mu|} \cap S^1_{|\mu'|}.$$

□

Proof. It is obvious that x is ANT if and only if $< v, A^{2k}(x) >$ and $< v, A^{2k+1}(x) >$ are both positive for k large. Now, reasoning as in the proof of Theorem 4.1, but using Propositions 4.3 and 4.4 instead of Proposition 4.2, we obtain that $< v, A^{2k}(x) >$ is positive for k large if and only if x belongs to $S^0_{|\mu|}$ for some $\mu \in |Spec(A)^*|$, and that $< v, A^{2k+1}(x) >$ is positive for k large if and only if x belongs to $S^1_{|\mu'|}$ for some $\mu' \in |Spec(A)^*|$. The result follows. □

The following two examples illustrate how Theorem 4.2 applies to different cases.

Example 4.2. (Running example) *Consider the program $P(A, v)$ depicted as follows:*

(i) Pseudo code:

```
while(2x+3y-z>0){
   x:=15x+18y-8z+6s-5t;
   y:=5x+3y+z-t-3s;
   z:=-4y+5z-4s-2t;
   s:=-43x-46y+17z-14s+15t;
   t:=26x+30y-12z+8s-10t;}
```

(ii) Associated matrices:

$$A = \begin{pmatrix} 15 & 18 & -8 & 6 & -5 \\ 5 & 3 & 1 & -1 & -3 \\ 0 & -4 & 5 & -4 & -2 \\ -43 & -46 & 17 & -14 & 15 \\ 26 & 30 & -12 & 8 & -10 \end{pmatrix}, \; v = \begin{pmatrix} 1 \\ 1 \\ 2 \\ 3 \\ 1 \end{pmatrix}.$$

Step 1*: Diagonalization of the matrix A:*

$$P = \begin{pmatrix} 2 & 1 & -1 & 1 & 1 \\ -1 & 0 & 2 & 0 & -1 \\ -2 & 0 & 2 & -1 & -2 \\ -4 & -1 & 0 & -2 & -1 \\ 2 & 2 & 1 & 2 & 1 \end{pmatrix}, D = \begin{pmatrix} -3 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{pmatrix} \text{ and } P^{-1} = \begin{pmatrix} -3 & -3 & 1 & -1 & 1 \\ -1 & -2 & 1 & 0 & 1 \\ -5 & -4 & 1 & -1 & 2 \\ 10 & 10 & -3 & 2 & -4 \\ -7 & -6 & 1 & -1 & 3 \end{pmatrix}.$$

We obtain the following eigenvectors written using our notation: $e_{0,1} = (-1, 2, 2, 0, 1)^\top$, $e_{1,1} = (1, 0, -1, -2, 2)^\top$, $e_{2,1} = (1, -1, -2, -1, 1)^\top$, $e_{-1,1} = (1, 0, 0, -1, 2)^\top$ *and* $e_{-3,1} = (2, -1, -2, -4, 2)^\top$.

Step 2*: Computing S_μ for all positive $\mu \in Spec(A)^*$:*

- *Our algorithm first computes the coefficients $c_{\lambda,i}$. We obtain :* $c_{0,1} = 6$, $c_{1,1} = -5$, $c_{2,1} = -6$, $c_{-1,1} = 0$ *and* $c_{-3,1} = -13$.
- *Then, our algorithm computes the coefficients of the decomposition of the initial variable values in B. They are those of the column vector $P^{-1} \cdot u$ in B_c where $u = (u_1, u_2, u_3, u_4, u_5)^\top$ is the vector encoding the initial variable values.*

$$P^{-1}. \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{pmatrix} = \begin{pmatrix} -3u_1 - 3u_2 + u_3 - u_4 + u_5 \\ -u_1 - 2u_2 + u_3 + u_5 \\ -5u_1 - 4u_2 + u_3 - u_4 + 2u_5 \\ 10u_1 + 10u_2 - 3u_3 + 2u_4 - 4u_5 \\ -7u_1 - 6u_2 + u_3 - u_4 + 3u_5 \end{pmatrix} = \begin{pmatrix} x_{-3,1} \\ x_{-1,1} \\ x_{0,1} \\ x_{1,1} \\ x_{2,1} \end{pmatrix}.$$

Now, the algorithm obtains all the non-zero terms appearing in the definition of $S^0_{|\lambda|}$ and $S^1_{|\lambda|}$:

$$< v, x_{|1|} + x_{-|1|} >= c_{1,1}x_{1,1} = -5(10u_1 + 10u_2 - 3u_3 + 2u_4 - 4u_5)$$
$$< v, x_{|2|} + x_{-|2|} >= c_{2,1}x_{2,1} = -6(-7u_1 - 6u_2 + u_3 - u_4 + 3u_5)$$

$< v, x_{|-3|} + x_{-|-3|} >= c_{-3,1}x_{-3,1} = -13(-3u_1 - 3u_2 + u_3 - u_4 + u_5)$
$< v, x_{|-3|} - x_{-|-3|} >= -c_{-3,1}x_{-3,1} = 13(-3u_1 - 3u_2 + u_3 - u_4 + u_5)$
All the sets $S^0_{|\lambda|}$ and $S^1_{|\lambda|}$ can now be computed exactly:

$$\begin{aligned}
S^0_{|1|} &= \{u \in E \,|\, 5(10u_1 + 10u_2 - 3u_3 + 2u_4 - 4u_5) > 0 \ \wedge \ -6(-7u_1 - 6u_2 + u_3 - u_4 + 3u_5) = 0 \ \wedge \\
&\qquad - 13(-3u_1 - 3u_2 + u_3 - u_4 + u_5) = 0\};\\
S^1_{|1|} &= \{u \in E | 5(10u_1 + 10u_2 - 3u_3 + 2u_4 - 4u_5) > 0 \ \wedge \ -6(-7u_1 - 6u_2 + u_3 - u_4 + 3u_5) = 0 \ \wedge \\
&\qquad 13(-3u_1 - 3u_2 + u_3 - u_4 + u_5) = 0\};\\
S^0_{|2|} &= \{u \in E| \ - 6(-7u_1 - 6u_2 + u_3 - u_4 + 3u_5) > 0 \ \wedge \ -13(-3u_1 - 3u_2 + u_3 - u_4 + u_5) = 0\};\\
S^0_{|2|} &= \{u \in E| \ - 6(-7u_1 - 6u_2 + u_3 - u_4 + 3u_5) > 0 \ \wedge \ -13(-3u_1 - 3u_2 + u_3 - u_4 + u_5) = 0\};\\
S^1_{|2|} &= \{u \in E| \ - 6(-7u_1 - 6u_2 + u_3 - u_4 + 3u_5) > 0 \ \wedge \ 13(-3u_1 - 3u_2 + u_3 - u_4 + u_5) = 0\};\\
S^0_{|-3|} &= \{u \in E| \ - 13(-3u_1 - 3u_2 + u_3 - u_4 + u_5) > 0\};\\
S^1_{|-3|} &= \{u \in E| \ 13(-3u_1 - 3u_2 + u_3 - u_4 + u_5) > 0\}.
\end{aligned}$$

Step 3*: We apply Theorem 4.2 to generate the ANT Locus:*
The algorithm computes the following intersections: $S^0_1 \cap S^1_1$, $S^0_1 \cap S^1_2$, $S^0_1 \cap S^1_3$, $S^0_2 \cap S^1_1$, $S^0_2 \cap S^1_2$, $S^0_2 \cap S^1_3$, $S^0_3 \cap S^1_1$, $S^0_3 \cap S^1_2$ and $S^0_3 \cap S^1_1$.
In fact, the previous computational step already allows our algorithm to perform implicit simplifications. For instance, it appears that $S^0_{|1|} = S^1_{|1|}$, $S^0_{|1|} = S^1_{|1|} = S^0_{|-1|} = S^1_{|-1|}$, $S^0_{|2|} = S^1_{|2|}$ and that $S^0_{|-3|} \cap S^1_{|-3|}$ is the empty set. According to Theorem 4.2, the ANT locus reduces to the following semi-linear space:

$$\begin{aligned}
\{u = (u_1, u_2, u_3, u_4, u_5)^\top \in E \ | \ & - 10u_1 - 10u_2 + 3u_3 - 2u_4 + 4u_5 > 0 \ \wedge \\
& - 7u_1 - 6u_2 + u_3 - u_4 + 3u_5 = 0 \ \wedge \\
& - 3u_1 - 3u_2 + u_3 - u_4 + u_5 = 0\} \ \bigcup \\
\{u = (u_1, u_2, u_3, u_4, u_5)^\top \in E \ | \ & - 10u_1 - 10u_2 + 3u_3 - 2u_4 + 4u_5 > 0 \ \wedge \\
& - 3u_1 - 3u_2 + u_3 - u_4 + u_5) > 0 \ \wedge \\
& - 7u_1 - 6u_2 + u_3 - u_4 + 3u_5 = 0\} \ \bigcup \\
\{u = (u_1, u_2, u_3, u_4, u_5)^\top \in E \ | \ & 7u_1 + 6u_2 - u_3 + u_4 - 3u_5 > 0 \ \wedge \\
& - 3u_1 - 3u_2 + u_3 - u_4 + u_5 = 0\}. \quad \square
\end{aligned}$$

5 Discussions

This work is complementary to our previous work [19], which dealt with termination analysis. In [19] we first prove a sufficient condition for the termination of homogeneous linear programs. This statement was conjectured in the important work of [21], where the first attempt to prove this result contains non trivial mistakes. As we shown in [19], the actual proof of this sufficient condition requires expertise in several independent mathematical fields. Also, the necessary condition proposed in [21] does not apply as expected in practice. We then went to generalize the previous result and, to the best of our knowledge, we presented the *first necessary and sufficient condition* (NSC, for short) for the termination of linear programs. In fact, this NSC exhibits a complete decidability result for the class of linear programs on all initial values. Moreover, departing from this NSC, we showed the scalability of these approaches by demonstrating that one can directly extract a sound and complete computational method, running in polynomial time complexity, to determine termination or nontermination for linear programs. On the other hand, all other related and previous works mentioned in this paper do not provide any techniques capable of generating automatically the set of initial input variable values for which a loop does not terminate. The main contributions of this paper remain on a sound and complete computational method to compute the set of input variable values for which the programs do not terminate. The overall time complexity of our algorithm is also of order $\mathcal{O}(n^3)$. As can be seen, the main results, *i.e.*, Theorems 4.1 and 4.2, provide us with a direct symbolic representation of the ANT set. Even if those theorems are rigorously stated and proofs are quite technical, they are really easy to apply: we only need to compute the explicit terms $S^0_{|\mu|}$ and $S^1_{|\mu'|}$ in order to directly obtain a formula representing exactly and symbolically the ANT set. In a same manner, we extended this techniques to linear program not necessarily diagonalizable and we obtained similar theoretical and practical results. As their associated proofs are more technical, they would required more space to be fully expressed and we left them for an ensuing report. In other more recent work on termination static analysis for programs over the rationals or the integers with several conditional linear inequalities, we also show that the notion ANT remains central.

6 Conclusion

We presented the new notion of *asymptotically non-terminant initial variable values* for linear programs. Considering a linear diagonalizable program, our theoretical results provided us with sound, complete and fast computational methods allowing the automated generation of the sets of all asymptotically non-terminant initial variable values, represented symbolically and exactly by a semi-linear space, e.g., conjunctions and disjunctions of linear equalities and inequalities. Also, by taking the complementary set of the semi-linear set of ANT initial variable values, we obtain a precise under-approximation of the set of terminant initial variable values for the (non -terminant) program. Actually, this type of method can be vastly generalized, to tackle the termination and non-termination problem of linear programs not necessarily diagonalizable, with more than one conditional linear inequality, on rational or integer initial values, for instance. We leave this investigation for an ensuing report.

References

[1] Amir M. Ben-Amram and Samir Genaim. On the linear ranking problem for integer linear-constraint loops. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 51–62, New York, NY, USA, 2013. ACM.

[2] Amir M. Ben-Amram, Samir Genaim, and Abu Naser Masud. On the termination of integer loops. In *VMCAI*, pages 72–87, 2012.

[3] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In *In CAV*, pages 491–504. Springer, 2005.

[4] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination analysis of integer linear loops. In *In CONCUR*, pages 488–502. Springer-Verlag, 2005.

[5] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination of polynomial programs. In *In VMCAI'2005: Verification, Model Checking, and Abstract Interpretation, volume 3385 of LNCS*, pages 113–129. Springer, 2005.

[6] Mark Braverman. Termination of integer linear programs. In *In Proc. CAV06, LNCS 4144*, pages 372–385. Springer, 2006.

[7] Hong Yi Chen, Shaked Flur, and Supratik Mukhopadhyay. Termination proofs for linear simple loops. In *Proceedings of the 19th international conference on Static Analysis*, SAS'12, pages 422–438, Berlin, Heidelberg, 2012. Springer-Verlag.

[8] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, Cambridge, MA, 2000.

[9] Michael Colón and Henny Sipma. Synthesis of linear ranking functions. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 67–81, London, UK, 2001. Springer-Verlag.

[10] Michael A. Colón and Henny B. Sipma. Practical methods for proving program termination. In *In CAV2002: Computer Aided Verification, volume 2404 of LNCS*, pages 442–454. Springer, 2002.

[11] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. *SIGPLAN Not.*, 41(6):415–426, June 2006.

[12] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[13] Patrick Cousot and Radhia Cousot. An abstract interpretation framework for termination. *SIGPLAN Not.*, 47(1):245–258, January 2012.

[14] Dennis Dams, Rob Gerth, and Orna Grumberg. A heuristic for the automatic generation of ranking functions. In *Workshop on Advances in Verification*, pages 1–8, 2000.

[15] Zohar Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

[16] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, pages 239–251, 2004.

[17] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th International Symposium in Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[18] Rachid Rebiha, Nadir Matringe, and Arnaldo V. Moura. A complete approach for termination analysis of linear programs. Technical Report IC-13-08, Institute of Computing, University of Campinas, February 2013.

[19] Rachid Rebiha, Nadir Matringe, and Arnaldo V. Moura. Necessary and sufficient condition for termination of linear programs. Technical Report IC-13-07, Institute of Computing, University of Campinas, February 2013.

[20] Henny B. Sipma, Tomás E. Uribe, and Zohar Manna. Deductive model checking. *Form. Methods Syst. Des.*, 15(1):49–74, July 1999.

[21] Ashish Tiwari. Termination of linear programs. In Rajeev Alur and Doron Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA*, volume 3114 of *Lecture Notes in Computer Science*, pages 70–82. Springer, 2004.

Data Conversion Method between a Natural Number and a Binary Tree for an Inductive Proof and Its Application

Kazuko Takahashi[1], Shizuo Yoshimaru[1*]
and Mizuki Goto[1]

[1] School of Science and Technology, Kwansei Gakuin University
`ktaka@kwansei.ac.jp`
[2] School of Science and Technology, Kwansei Gakuin University
`shizuo.yoshimaru@gmail.com`
[3] School of Science and Technology, Kwansei Gakuin University
`bub85144@kwansei.ac.jp`

Abstract

This paper presents modeling of a binary tree that represents a natural number and gives an inductive proof for its properties using theorem provers. We define a function for converting data from a natural number into a binary tree and give an inductive proof for its well-definedness. We formalize this method, develop a computational model based on it, and apply it to an electronic cash protocol. We also define the payment function on the binary tree and go on to prove the divisibility of electronic cash using the theorem provers Isabelle/HOL and Coq, respectively. Furthermore, we discuss the effectiveness of this method.

1 Introduction

Theorem proving is an important technique to provide a certified system or to ensure bug-free programming. Several theorem provers, which may be called proof assistants, have been developed, including ACL2 [11], PVS [19], Isabelle/HOL [16], Coq [4], Agda [17], and so on. These tools help users to develop formal proofs, either interactively or semi-automatically, typically using induction as the proof strategy. There are numerous applications for such provers. There are relatively few practical applications of these tools in most fields compared to their use in pure mathematics, although there are some outstanding results in microprocessor design [10], C compilers [14], operating system kernels [12], security protocols [1], and secure card systems [3, 13]. To use these provers successfully, it is necessary to construct a suitable model and then select an appropriate proof strategy. In practical applications it may be difficult to form a natural inductive model, and so it may be necessary to build a data structure, which can be difficult for these provers to handle.

We have previously attempted to prove the divisibility of an electronic cash protocol [21]. In this experience, we have encountered difficulties in proving the properties regarding the function that converts a natural number to a binary tree. The binary tree is a complete binary tree, where a Boolean value is attached to each node, and the value of which is defined as a sum of the value of the left and right subtrees. The problem can be reduced to the fact that there exists a function that cannot be defined in a primitive recursive form on an inductively defined data structure. Because a natural number has a linear structure with only one successor, whereas a

[*]Currently, Nochu Information System Co.,LTD.

binary tree has a branching structure with two successors at each branching point, the induction schemes are different, which complicates the proof.

To solve this problem, we introduced a bit sequence as an intermediate data structure, and defined the function from a natural number to a binary tree as a composition of two recursive functions via a bit sequence. Specifically, the first function describes the mapping from a natural number to an intermediate bit sequence and the second function is the mapping from the bit sequence to the binary tree. We formalized this model and associated proof method, and applied it to the divisibility of an electronic cash protocol using Isabelle/HOL. However, this previous work had several drawbacks. First, the model was complicated, because of an intricate labeling for the nodes of a tree; a simpler and more natural model for this specific binary tree has subsequently been identified. Second, the proof was incomplete, as one of the lemmas that relates the bit sequence to the binary tree was left as an axiom. Third, the effectiveness of the method was not discussed; we did not know whether this method could be applied similarly with theorem provers other than Isabelle/HOL, or whether a large number of modifications would be required. In this paper, we provide a simplified model, give complete proofs of the divisibility using the theorem provers Isabelle/HOL and Coq, and discuss the effectiveness of the method.

The rest of this paper is organized as follows. In Section 2, we describe in detail the problems of data conversion and our solution. In Sections 3 and 4, we present a formalization and an inductive proof of the divisibility of an electronic cash protocol. In Section 5, we provide a discussion, and in Section 6, we present our conclusions.

2 Data Conversion Formalization

2.1 Problem

First, we illustrate the problem of converting between a natural number and a binary tree, which are data structures with different induction schemes.

Let NAT and BTREE be a natural number and a binary tree, respectively. NAT is inductively defined with one successor. BTREE is inductively defined in such a form that there are two successors, i.e.,

```
nat_p(0).
nat_p(n) => nat_p(Suc(n)).

tree_p(Tip).
tree_p(lt) & tree_p(rt) => tree_p(Node(_,lt,rt)).
```

where Suc denotes the successor function; $Node$ is a constructor for a tree; '_' denotes an anonymous variable that corresponds to a parent node; lt and rt are the left and right subtrees, respectively; and Tip is a leaf node. The symbols & and => denote conjunction and implication, respectively.

Consider data conversion between NAT and BTREE. Let f be a function that maps from NAT to BTREE and g be a function that maps from BTREE to NAT. We assume that these functions are defined inductively in the following form,

```
f: NAT --> BTREE
f(Suc(n)) = c1(f(n)).
f(0)      = Tip.
```

```
g: BTREE --> NAT
g(Node(_,lt,rt)) = c2(g(lt), g(rt)).
g(Tip)           = 0.
```

where $c1$ is a function from BTREE to BTREE, and $c2$ is a function from NAT× NAT to NAT.

Consider proving the following relationship between f and g,

$$g(f(n)) = n.$$

We use the following induction scheme IS on NAT to prove it.

$$\forall n.(\ (\forall n'.n' < n \Longrightarrow g(f(n')) = n') \Longrightarrow g(f(n)) = n\ ) \qquad \text{[IS]}$$

The proof proceeds by rewriting $g(f(n))$ in succession as follows:

$$\begin{aligned} g(f(n)) &= g(f(c2(n1, n2))) \\ &= g(Node(\_, f(n1), f(n2))) \\ &= c2(g(f(n1)), g(f(n2))) \\ &= c2(n1, n2) \end{aligned}$$

where $c2(n1, n2) = n$ for a certain $n1, n2 < n$.

The induction scheme IS is used to proceed from the third line to the fourth line. To succeed in this proof, there are two problems to be solved. The first is the progression from the first line to the second line: we have to prove the equality $f(c2(n1, n2)) = Node(\_, f(n1), f(n2))$. This problem is how to divide a natural number into two parts suitably. The second, and arguably more challenging problem, is that we cannot find $c1$ that defines f in a recursive form naturally. Consider the function f, which maps a natural number to a binary tree. The forms of the data structures $f(n)$ and $f(Suc(n))$ are considerably different, and the differences depend on the values of n. For example, compare the operation of adding a leaf node labeled "True" to the binary tree shown in Figure 1. In this figure, (a) shows the operation of adding a node to the binary tree that has two leaf nodes labeled "True," and (b) shows the operation of adding a node to the binary tree that has three leaf nodes labeled "True." These operations cannot be represented uniformly, therefore $f(n)$ cannot be uniformly defined for all n. Furthermore, even if f is defined, the proof over the properties on f is not straightforward.

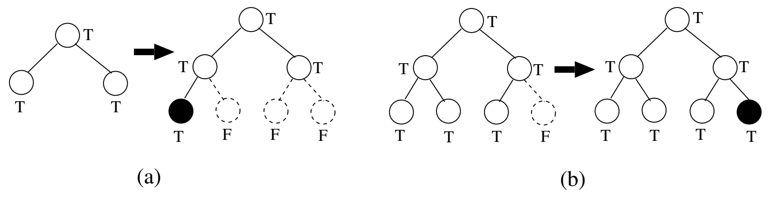


Figure 1: Operation on adding a "True" node to a binary tree

On the other hand, the definition of g is simpler because it is a conversion from a data type that is not totally-ordered to one that is totally-ordered.

2.2 Solution

To solve the problem shown above, we introduce the bit sequence BS as an intermediate data structure between NAT and BTREE.

2.2.1 Conversion from BS to BTREE

First, we describe the conversion from BS to BTREE. We choose a specific binary tree used in an electronic cash protocol. This tree is a complete binary tree and a Boolean value is attached to each node, the value of which is defined as a sum of the values of its left and right subtrees.

Let $b_0 b_1 \ldots b_n$ be a sequence of Boolean values that corresponds to a natural number k. Then $k = b'_0 \cdot 2^n + b'_1 \cdot 2^{n-1} + \ldots + b'_n \cdot 2^0$ where $b'_i = 1/0$ for $b_i = True/False$ $(1 \leq i \leq n)$, respectively. We encode $b'_0 \cdot 2^n$ as the left tree and the remainder as the right tree. The function from BS to BTREE is defined as follows:

```
bs_to_btree() = (Tip,false)
bs_to_btree(b#bs) = (if b
    then Node(true, create_btree(true,length(bs)), bs_to_btree(bs) )
    else Node(true, bs_to_btree(bs), create_btree(false,length(bs)) )
    )
```

where # is an operator combining the head and tail of the list; *Tip* indicates a leaf node; the function *create_btree*(*bool*, *nat*) creates a binary tree of height *nat* in which all nodes are labeled with *bool*; and *length*(*list*) denotes the length of *list* [1]. Intuitively, when scanning the data from the head of BS, descending by one bit corresponds to descending one subtree in BTREE. The induction scheme for the bit sequence is that if some property holds on a bit sequence *bs*, it also holds on *b#bs*. If $b = True$ (i.e., $k \geq 2^n$), then all nodes in the left subtree are labeled $True$; this is referred to as a $full\_tree$. However, if $b = False$ (i.e., $k < 2^n$), then none of the nodes in the right tree are labeled $True$; this is referred to as an *empty_tree*. Thus, the data types BS and BTREE are matched in their induction schemes.

Let a bit sequence be represented as a list, the elements of which are either $True$ or $False$. Figure 2 shows the conversion of the bit sequence $[True, False, True]$ into a binary tree. In this figure, the black trees are full trees, the white trees are empty trees, and the dotted trees are the others.

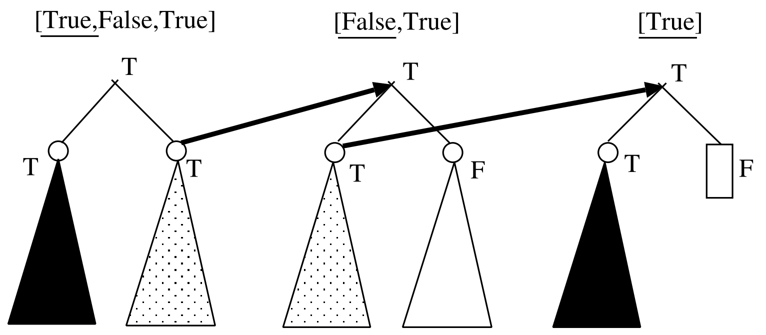


Figure 2: The mapping between a bit sequence and a binary tree

Because the binary tree corresponding to $[True, False, True]$ is not an empty tree, the root node is labeled $True$. Because the first bit in the sequence is $True$, the left tree is a full tree. The tree to the right of it is the tree corresponding to the remaining bit sequence $[False, True]$ after the first bit has been extracted. Now consider the coding of $[False, True]$; because the

[1]We can define a mirror tree in which left and right subtrees are exchanged.

first bit of the sequence is $False$, the right tree must be an empty tree, and the left tree is the tree corresponding to the remaining bit sequence. Then we consider the coding of $[True]$; because the first bit of the bit sequence is $True$, the left tree is a full tree, and the right tree is the binary tree corresponding to the remaining bit sequence. Finally, we obtain a Tip for [ ].

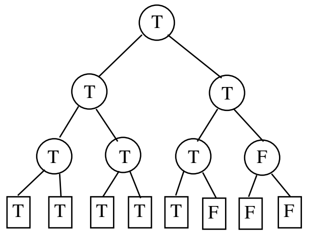


Figure 3: The binary tree for $[True, False, True]$

The binary tree obtained from $[True, False, True]$ is thus as follows (Figure 3):

```
( True,
  ( True, ( True,True,True ),  ( True,True,True ) ),
  ( True, ( True,True,False ), ( False,False,False ) )
)
```

The function *bs_to_btree* can give an inductive definition that provides for the division of n into $n1$ and $n2$, corresponding to the left and right subtrees, respectively, in the proof process for IS in the previous subsection.

2.2.2 Conversion from NAT to BS

Next, we give a definition for the conversion from NAT to BS. We show two different models: one using a general function and one using a primitive recursive function.

First, we define the general function *naive_nat_to_bs*. This is defined in an iterative form and determines the value of a a bit sequence from the least-significant bit as follows:

```
naive_nat_to_bs(0) = [False]
naive_nat_to_bs(1) = [True]
naive_nat_to_bs(n) = (naive_nat_to_bs(div(n,2))) @ ([mod(n,2) = 1])
```

where @ is the concatenation operator for lists, $div(n, 2)$ and $mod(n, 2)$ return the result of division of n by 2 and its remainder, respectively.

Second, we define the primitive recursive function *nat_to_bs*. This is defined in a tail-recursive form and determines the value of a bit sequence from the most-significant bit. The definition is more complex than that of the iterative form.

```
nat_to_bs(0) = [false]
nat_to_bs(Suc(n)) = calc(lg(Suc(n)),Suc(n))

calc(0,m) = [true]
calc(Suc(n),m) = (if 2^n <= m) then True#(calc(n,m-2^(Suc(n))))
                         else False#(calc(n,m))

lg(n) = (if n<=1 then 0
             else Suc(lg (div(n,2))))
```

where lg is a key function that returns the place number of the bit sequence corresponding to a given argument, i.e., the natural number m that satisfies $m \leq \log\ n < m + 1$. The following table shows the relationship of a natural number nat, a bit sequence bs, and the value of lg. Note that when n increases by 1, the location in the bit sequence, which is equal to the height of the tree, increases logarithmically.

nat	bs	lg
1	1	0
2	10	1
3	11	1
4	100	2
5	101	2
6	110	2
7	111	2
8	1000	3
9	1001	3
	⋮	

2.2.3 Conversion from NAT to BTREE

Finally, *nat_to_btree* is defined as the composition of the two functions *naive_nat_to_bs* and *bs_to_btree* or the composition of *nat_to_bs* and *bs_to_btree*. *nat_to_bs* and *bs_to_btree* are primitive recursive functions, whereas *naive_nat_to_bs* is a recursive function.

The definition of *nat_to_bs* is more suitable to an induction scheme than that of *naive_nat_to_bs*, because induction is typically applied from the head to the tail.

In the next two sections, we apply this formalization to the electronic cash protocol defined in [18] and prove its divisibility.

3 Modeling of an Electronic Cash Protocol

3.1 Ideal Electronic Cash

An electronic cash (e-cash) protocol is, in general, a combination of cryptography techniques, such as zero-knowledge proof and public key encryption. We consider the ideal e-cash protocol proposed by Okamoto [18]. In this protocol, a coin of some monetary value is encoded as a binary tree, and a payment function is defined over it. This binary tree approach makes e-cash efficient and unlinkable, and is used in many divisible e-cash schemes [7, 8, 15, 18]. We formalize this protocol and prove its divisibility on a data level, i.e., a user can spend a coin in several separate transactions by dividing its value without overspending if and only if a payment function satisfies the payment rules. This is one of the properties that an ideal e-cash protocol should satisfy. We define the data structure of *money* and define two primitive recursive functions of *money_amount* and *pay* on *money*. The definitions are shown using Coq code in this section.

3.2 Definition of *money*

A coin is represented as a binary tree called *money*. Each node of the tree represents a certain denomination. The root node is assigned the monetary value of the money, and the values of all other nodes are defined as half the value of their parent nodes.

Money is defined as an inductive function. It is a binary tree, where nodes are labeled with Boolean values. The label $True$ means that a node is usable, while $False$ means that it is not. For a given Boolean value b and a natural number n, we can create *money* in which all of the nodes have the label b.

```
Inductive money : Type :=
| Tip  : bool -> money
| Node : bool -> money -> money -> money.

Fixpoint create_money(b : bool)(n : nat) : money :=
 match n with
| 0 => Tip b
| S n' => Node b (create_money b n') (create_money b n')
 end.
```

3.3 Creation of *money* from a natural number

cash is a function that creates *money* corresponding to a given natural number that is defined as a composition of *bs_to_money* and *naive_nat_to_bs* (or *nat_to_bs*). *bs_to_money* can be defined in a manner similar to that of *bs_to_btree*.

```
Definition cash(n : nat) : money := bs_to_money (naive_nat_to_bs n).
```

Note that *cash* is the money that satisfies a specific condition, whereas type *money* allows any complete binary tree whose nodes are labeled as Boolean values. For example, $Node(false, Tip(true), Tip(true))$ is an element of *money* but not a result of a function *cash*.

3.4 Calculation of the amount of *money*

The function *money_amount* computes the amount of money that can be used. If the root node is labeled $True$, the amount of the tree is the sum of that of the left tree and the right tree; otherwise, the amount is 0.

```
Fixpoint money_amount(m : money) : nat :=
 match m with
| Tip true => 1
| Tip false => 0
| Node true l r => money_amount l + money_amount r
| Node false _ _ => 0
 end.
```

3.5 Payment

Payment rules are set in [18] as follows: when we spend some amount from the coin, we search for a node (or combination of nodes) whose value is equal to the payment value, and then cancel these nodes; at the same time, all of the ancestors and all of the descendants are also canceled. Overspending is prevented if and only if these rules are satisfied.

The function *pay* corresponds to payment according to the payment rules. When we pay n from *money*, where n is less than or equal to the amount of *money*, then we pay all of n from the left tree, and the right tree remains as it is if the amount of the left tree is more than n.

Otherwise, we exhaust the left tree in the payment and pay the remainder from the right tree. For example, $money\_amount(pay(cash(13), 4) = 9$.

In the following code, eq_nat_dec and lt_dec are functions on NAT that denote '=' and '<', respectively. $change\_false$ is a function that changes the label of a node to $False$.

```
Fixpoint pay(m : money)(n : nat) : money :=
 match m with
| Tip true => Tip (if eq_nat_dec n 0 then true else false)
| Node true l r =>
if lt_dec (money_amount l) n
then Node true (change_false l) (pay r (n - money_amount l))
else Node true (pay l n) r
| _ => m
 end.
```

4 Proof of Properties

We prove three properties on the divisibility of the e-cash protocol. On proving these properties inductively, we apply the method proposed in Section 2.

In this section, we only show the outline of the proof, focusing on how induction is applied. The complete Coq proof is shown in the Appendix.

First, the distribution property over $money\_amount$ holds.

$$\begin{aligned} &money\_amount(Node(\_, left, right)) = \\ &\qquad money\_amount(left) + money\_amount(right) \quad \cdots (1) \end{aligned}$$

This can readily be proved. We rewrite the left side of the given equation.

4.1 Well-definedness of $cash$

The monetary value of $money$ created by $cash$ from a natural number is equal to that value. This property is represented as follows,

$$\forall\ n.\ (money\_amount\ (cash(n)) = n)$$

and is proved using the properties of the bit sequence. Below, we show a case in which the first bit b of the sequence is $True$. When it is $False$, the proof proceeds in the same manner.

$$\begin{aligned} &money\_amount(cash(n)) \\ &= money\_amount(bs\_to\_money(b\#bs)) && \cdots (2) \\ &= money\_amount(\ Node(true, \\ &\qquad\qquad bs\_to\_money(bs), create\_money(true, length(bs)\ ) && \cdots (3) \\ &= money\_amount(bs\_to\_money(bs)) + \\ &\qquad money\_amount(create\_money(true, (length(bs)))) && \cdots (4) \end{aligned}$$

Formula (2) is obtained by unfolding $cash$, where $nat\_to\_bs(n) = b\#bs$, formula (3) is obtained by unfolding bs_to_money, and formula (4) is obtained by the distribution property of $money\_amount$ (1).

The first term of formula (4) is transformed as follows.

$$\begin{aligned}
&money\_amount(bs\_to\_money(bs)) \\
&= money\_amount(bs\_to\_money(nat\_to\_bs(n - 2^{lg\ n}))) && \cdots (5) \\
&= money\_amount(cash(n - 2^{lgn})) && \cdots (6)
\end{aligned}$$

Formula (5) is obtained using case split tactics on n, property of lg and several other tactics. Formula (6) is obtained by unfolding $cash$.

The second term of formula (4) is transformed as follows:

$$\begin{aligned}
&money\text{-}amount(create\_money(true, length(bs))) \\
&= money\_amount(cash(2^{lgn})) && \ldots (7)
\end{aligned}$$

Here we use the following induction scheme of NAT.

$$\begin{aligned}
&\text{if } \forall\ k;\ k < n,\ money\_amount(cash(k))\ =\ k, \\
&\text{then } money\_mount(cash(n))\ =\ n \text{ holds.}
\end{aligned}$$

We can prove $2^{lg\ n} \leq n$ and $n - 2^{lg\ n} < n$. If $2^{lg\ n} < n$, we apply this type of induction to both formulas (6) and (7), and so formula (4) is finally transformed into the following form:

$$\begin{aligned}
&money\_amount(cash(2^{lg\ n})) + money\_amount(cash(n - 2^{lg\ n})) \\
&= 2^{lg\ n} + (n - 2^{lg\ n}) \\
&= n
\end{aligned}$$

In case $2^{lg\ n} = n$, the proof is simpler without using induction.

4.2 Well-definedness of *pay*

The amount remaining after payment is the difference between the original value and the payment value. This property is represented as follows [2]:

$$\forall\ n.\forall\ m.\ (money\_amount(pay(cash(n), m)) = n - m)$$

This is derived from the following lemma:

$$\forall\ c.\forall\ n.\ (money\text{-}amount(pay(c, n)) = money\text{-}amount(c) - n$$

which is proved as follows. When we pay n from c, if n does not exceed the amount of c, we pay $m1$ from the left subtree as far as possible, and pay the remainder $m2$ from the right tree. $m1$ and $m2$ are determined as follows: if $n < money\_amount(left)$, then $m1 = n$ and $m2 = 0$; otherwise, $m1 = money\_amount(left)$ and $m2 = n - money\_amount(left)$. This is proved using induction. Below, we show an inductive case, because the proof is simple for the base case.

$$\begin{aligned}
&money\_amount(pay(c, m1 + m2)) \\
&= money\_amount(pay(Node(\_, left, right), m1 + m2)) && \cdots (8) \\
&= money\_amount(pay(left, m1)) + money\_amount(pay(right, m2)) && \cdots (9) \\
&= money\_amount(left) - m1 + money\_amount(right)\ -\ m2 && \cdots (10) \\
&= (money\_amount(left)\ +\ money\_amount(right))\ -\ (m1\ +\ m2) \\
&= money\_amount(Node(\_, left, right))\ -\ (m1 + m2) && \cdots (11) \\
&= money\_amount(c)\ -\ (m1 + m2)
\end{aligned}$$

[2] Note that $n - m = 0$ when $m > n$.

Formula (8) is obtained by expanding c. Formula (9) is obtained by the distribution property of *money_amount* and the property of payment on *money*. Formula (10) is obtained by applying induction to *money*. Formula (11) is obtained from the distribution property of *money_amount*.

4.3 Divisibility

Given that ms is a list of values, each of which corresponds to a transaction, if a user pays from the head of this list in succession, then the remainder is the correct value. This property is represented as follows:

$$\begin{aligned} &\forall n.\forall ms.(\ \ n \geq listsum(ms) \\ &\Longrightarrow money\_amount\ (foldl(pay(cash(n), ms)))\ =\ n\ -\ listsum(ms)\ ) \end{aligned}$$

Here, $foldl$ and $listsum$ are the functions handling a list. Let ms be a list $[m1, \ldots, mk]$. $foldl(pay(c, ms))$ is rewritten in the following form:

$$(pay(\ \ldots\ (pay(c, m1),\ \ldots\ mk)$$

and $listsum(ms)$ is rewritten in the following form:

$$m1\ +\ \ldots\ +\ mk$$

This theorem can be proved using the result of the proofs for the above two properties of well-definedness.

5 Discussion

Modeling the e-cash protocol and proving the properties described in the previous sections were performed using the theorem provers Isabelle/HOL and Coq, respectively. First, we compare these two provers.

Both of them are interactive theorem-proving environments based on inductive theorem proving. The data types and functions are defined in recursive form, and the proof proceeds by connecting suitable tactics.

Isabelle/HOL has a more powerful engine for automatic proof than Coq. A proof may succeed simply by using the 'auto' command without connecting multiple lemmas in Isabelle/HOL, whereas a user must specify tactics manually in Coq. However, the proof procedure in Coq is easier to understand.

Both provers are based on typed logics and adopt higher-order functions. In Coq, type-checking is richer and proof-checking is reduced to type checking. It requires the user to prove the termination of a recursive function. Isabelle/HOL also requires the user to prove the termination of a recursive function, but has a stronger automatic mechanism, and user does not need to supply much input.

We developed a model for an e-cash protocol, and set out to prove three properties of the model using Isabelle/HOL, and subsequently translated the model into Coq. The translation was basically straightforward. The definition of a primitive recursive function using *primrec* in Isabelle/HOL was translated to *Fixpoint* in Coq; the definition of a recursive function using fun was translated to *Function*; the definition of a non-recursive function using $definition$ was translated to $Definition$. In addition, we must prove the termination of the function introduced via *Function*.

The tactics used in the proofs of two provers were quite different. Generally, more tactics are required in Coq; however, the proof in Coq provides a useful basis from which to make a proof in Isabelle/HOL. Actually, we completed the proof of the unproved lemma in Isabelle/HOL by referring to the proof in Coq. From this experience, it is expected that similar modeling with other provers is possible and that the existing proofs will be useful as a basis for forming proofs in those other provers. All definitions and proofs in Isabelle/HOL and Coq are shown in [22].

In the field of protocol verification, there are a number of works on the verification of security protocols using Isabelle/HOL (e.g.,[2]). However, they mainly proved security or safety of protocols. To the best of our knowledge, there exists no research on a proof that focuses on the divisibility of electronic cash protocols using a theorem-proving approach.

Bijection between the set of natural numbers and rooted trees has been discussed in several works [5, 6, 9]. In these works, prime decomposition of the natural numbers was used to construct the corresponding rooted tree. They used an incomplete tree, in which each prime number forms an individual branch. It appears to be impossible to define this bijection in an inductive manner. Attempts to provide the mechanical proof were not made, and the correctness of the methods has not been proved using theorem provers. On the other hand, we propose a method for automated theorem proving using a complete binary tree.

The data conversion method described here is applicable to conversion from an inductively defined data structure with one successor to one with n successors by introducing a list in which each element takes n values as an intermediate data structure instead of a bit sequence. Following this transformation, the proof can proceed in a similar manner to the one shown in this paper, although the number of lemmas would increase. Moreover, it can be extended to data conversion from data structures with m successors to data structures with n successors by composing two data-conversion functions, i.e., one from an inductively defined data structure with m successors to that with one successor, and then a second function from an inductively defined data structure with one successor to one with n successors.

6 Conclusion

We have described a function for converting data from a natural number to a binary tree, and have given an inductive proof for its well-definedness. We have described a method of introducing a bit sequence as an intermediate data structure to provide a model for inductively defined data structures with different induction schemes. We formalized this method, developed a computational model based on it, and applied it to an electronic cash protocol. We succeeded in proving the properties of divisibility of the protocol using the theorem provers Isabelle/HOL and Coq, and discussed the effectiveness of the method.

In future, we would like to investigate other functions and/or properties on such a binary tree that is handled here, and develop methods of their inductive proof.

References

[1] Arsac, W., Bella, G., Chantry, X. and Compagna, L. : *Multi-Attacker Protocol Validation*, J. of Automated Reasoning 46(3-4):353-388 (2011).

[2] Bella, G., Massacci, B. and Paulson, L. : *Verifying the SET Purchase Protocols*, J. of Automated Reasoning 36:5 37 (2006)

[3] Bella, G. : *Inductive Verification of Smart Card Protocols*, J. of Computer Security 11(1):87-132 (2003).

[4] Bertot, Y. and Castŕan, P. : *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*, Springer Verlag (1998).

[5] Beyer, T. and Hedetniemi, S. M. : *Constant Time Generation of Rooted Trees*, SIAM J. Comput., 9(4):706-712 (1980).

[6] Cappello, P. : *A New Bijection between Natural Numbers and Rooted Trees*, 4th SIAM Conference on Discrete Mathematics (1988).

[7] Chan, A., Frankel, Y. and Tsiounins, Y. : *Easy Come - Easy Go Divisible Cash*, EUROCRYPT98, pp. 561-575 (1998).

[8] Canard, S. and Gouget, A. : *Divisible E-Cash Systems Can Be Truly Anonymous*, EUROCRYPT2007, pp.482-497 (2007).

[9] Göbel, F. : *On a 1-1-Correspondence between Rooted Trees and Natural Numbers*, J. of Combinatorial Theory, Series B(29):141-143 (1980).

[10] Hardin, D. S. (ed): *Design and Verification of Microprocessor Systems for High-Assurance Applications*, Springer Verlag (2010).

[11] Kaufmann,M., Monolios, P. and Moore, J. S. : *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers (2000).

[12] Klein, G. et al. : *seL4: Formal Verification of an Operating-System Kernel*, Commun. ACM 53(6): 107-115 (2010).

[13] Kurita, T. and Nakatsugawa, Y. : *The Application of VDM to the Industrial Development of Firmware for a Smart Card IC Chip*, Int. J. of Software and Informatics 3(2-3):343-355 (2009).

[14] Leroy, X. *Formal Verification of a Realistic Compiler*, Communications of the ACM, 52(7):107-115 (2009).

[15] Nakanishi, T. and Sugiyama, Y. : *An Efficiently Improvement on an Unlinkable Divisible Electronic Cash System*, IEICE Trans. on Fundamentals, E85-A(19):2326-2335 (2002).

[16] Nipkow, T., Paulson, L. and Wenzel, M. : *Isabelle/HOL A Proof Assistant for Higher-Order Logic*, Springer Verlag (2002).

[17] Norell, U. : *Dependently Typed Programming in Agda*, Advanced Functional Programming 2008: pp.230-266 (2008).

[18] Okamoto, T. : *An Efficient Divisible Electronic Cash Scheme*, The proceedings of Crypto'95, pp.438-451 (1995).

[19] Owre,S., Rushby,J.M. and Shankar,N. : *PVS: A Prototype Verification System*, The proceedings of CADE-11, pp.748-752 (1992).

[20] Sprenger, C., Zurich, E. T. H., Basin, D., et al. : *Cryptographically Sound Theorem Proving* 19th IEEE Computer Security Foundations Workshop (2006).

[21] Takahashi, K. and Yoshimaru, S. : *Formalization of Data Conversion for Inductive Proof* Tunisia-Japan Workshop on Symbolic Computation in Software Science (SCSS 2009), pp.135-150 (2009).

[22] `http://ist.ksc.kwansei.ac.jp/~ktaka/EMONEY`

Appendix. Proof for E-cash protocol in Coq (tail-recursive version)

```
Require Import Omega List Arith Div2 Bool Recdef Wf_nat.

Theorem well_definedness_of_pay :
forall(m : money)(n : nat),
    n <= money_amount m
 -> money_amount(pay m n) = money_amount m - n.
Proof.
intros m.
induction m; intros n LNM; destruct b; simpl in *; try reflexivity.
 destruct (eq_nat_dec n 0).
  rewrite e; reflexivity.
  assert(n = 1) by omega.
  rewrite H; reflexivity.
 destruct (lt_dec (money_amount m1) n); simpl.
  rewrite IHm2; try omega.
  replace (money_amount (change_false m1)) with 0; try omega.
  destruct m1; simpl; reflexivity.
  rewrite IHm1; omega.
Qed.

Lemma bit_sequence_distribution :
forall(bs : list bool)(b : bool),
 money_amount(bs_to_money(bs ++ (b::nil))) =
        2 * money_amount(bs_to_money bs) + (if b then 1 else 0).
Proof.
intros bs b.
induction bs; simpl in *.
 destruct b; reflexivity.
 assert(forall(bl : bool),money_amount (create_money bl (length
        (bs ++ b :: nil))) =
  money_amount (create_money bl (length bs)) + money_amount
        (create_money bl (length bs))).
  intros; clear IHbs; destruct bl; induction bs; simpl; try omega.
 destruct a; simpl in *; rewrite IHbs; repeat rewrite plus_0_r;
        repeat rewrite plus_assoc; rewrite H; omega.
Qed.

Lemma one_reminder_div2 :
forall(n : nat),
 (2 * div2 n) + (if one_reminder n then 1 else 0) = n.
Proof.
intros n.
case_eq (one_reminder n); intros; induction n using lt_wf_ind.
destruct n; try discriminate.
destruct n; try reflexivity.
simpl in H.
simpl.
replace (S (div2 n + S (div2 n + 0) + 1))
with (S (S (2 * div2 n + 1))).
```

```
repeat f_equal.
rewrite H0; try reflexivity; try omega.
apply H.
simpl.
omega.
destruct n; try reflexivity.
destruct n; try discriminate.
simpl in H.
simpl.
replace (S (div2 n + S (div2 n + 0) + 0))
with (S (S (2 * div2 n + 0))).
repeat f_equal.
rewrite H0; try reflexivity; try omega.
apply H.
simpl.
omega.
Qed.

Theorem well_definedness_of_cash :
forall(n : nat),
 money_amount (cash n) = n.
Proof.
intros n.
induction n using lt_wf_ind.
destruct n; try (simpl; reflexivity).
destruct n; try (simpl; reflexivity).
unfold cash; rewrite bit_sequence_equation.
case_eq (one_reminder (S (S n))); intros;
 rewrite bit_sequence_distribution; unfold cash in H; rewrite H;
 try (apply lt_div2; apply lt_0_Sn);
 pose (one_reminder_div2 (S (S n)));
 rewrite H0 in e; apply e.
Qed.

Definition listsum(ns : list nat) : nat := fold_right plus 0 ns.

Definition payment_amount(m : money)(ns : list nat) : nat :=
        money_amount (fold_left pay ns m).

Theorem Divisibility.
forall(n : nat)(ns : list nat),
 listsum ns <= n ->
 payment_amount (cash n) ns = n - listsum ns.
Proof.
intros n ns LSM.
induction ns using rev_ind; simpl in *.
 rewrite <- minus_n_O.
 apply well_definedness_of_cash.
 unfold payment_amount in *.
 unfold listsum in *.
 rewrite fold_left_app;
 rewrite fold_right_app in *; simpl in *.
 rewrite plus_0_r in *.
 assert(fold_right plus 0 ns + x <= n).
  generalize n LSM.
  clear IHns LSM n.
  induction ns; intros; simpl in *.
   omega.
   assert(fold_right plus x ns <= n - a) by omega.
   apply IHns in H; omega.
 rewrite well_definedness_of_pay; rewrite IHns; simpl in *; try omega.
 replace (fold_right plus x ns) with ((fold_right plus 0 ns) + x).
 omega.
 clear IHns H LSM.
 induction ns; simpl.
  reflexivity.
  rewrite <- IHns; info omega.
Qed.
```

Computer Algebra Investigation of Known Primitive Triangle-Free Strongly Regular Graphs

Mikhail Klin and Matan Ziv-Av

Ben-Gurion University of the Negev
klin@cs.bgu.ac.il
matan@svgalib.org

Abstract

With the aid of computer algebra systems COCO and GAP with its packages we are investigating all seven known primitive triangle-free strongly regular graphs on 5, 10, 16, 50, 56, 77 and 100 vertices. These graphs are rank 3 graphs, having a rich automorphism group. The embeddings of each graph from this family to other ones are described, the automorphic equitable partitions are classified, all equitable partitions in the graphs on up to 50 vertices are enumerated. Basing on the reported computer aided results and using techniques of coherent configurations, a few new models of these graphs are suggested, which are relying on knowledge of just a small part of symmetries of a graph in consideration.

1 Introduction

This paper appears on the edge between computer algebra systems and algebraic graph theory (briefly AGT) and is mainly devoted to the computer aided investigation of the known primitive triangle free strongly regular graphs.

In fact, there are 7 such graphs, those on 5, 10, 16, 50, 56, 77 and 100 vertices. The largest one, denoted by $NL_2(10)$ is the universal graph for this family, in the sense that all seven graphs are induced subgraphs of $NL_2(10)$. We denote by $\mathfrak{F}$ the family of those 7 graphs.

The history of the discovery of the graphs from $\mathfrak{F}$, and in particular of $NL_2(10)$, is quite striking. Indeed, the graph $NL_2(10)$ was discovered twice: once in 1956 by Dale Mesner, and second time, independently, in 1968 by D. G. Higman and C. C. Sims. The details of this history are considered in [15], see also a very brief summary in Section 4.

The graphs on 5, 10 and 50 vertices form a very significant subfamily, consisting of the known Moore graphs. One more possible member of this series (see e.g. [16]) would have 3250 vertices; its existence is an open problem of a great significance. Moore graphs and their natural generalizations are objects of much interest in extremal graph theory; they serve as the best examples of optimal network topologies. Discovery of a new primitive triangle free strongly regular graph would be a high level sensation in modern mathematics.

This text is oriented towards an interdisciplinary audience, though its kernel is strictly related to a concrete kind of application of computer algebra in AGT.

Section 2 provides in compact form the necessary background about strongly regular graphs, see [11], [5] for more details, while in section 3 we briefly introduce the tools of computer algebra used in AGT. The seven graphs from the family $\mathfrak{F}$ are presented in section 4.

Our first main result is introduced in Section 5: full description of embeddings of (primitive and imprimitive) triangle free strongly regular graphs into the graphs in $\mathfrak{F}$.

The next topic of our investigation is related to equitable partitions with a special emphasis on automorphic equitable partitions, see definitions in Section 6. The search strategy is described in Section 7, though much more details may be found in [28]. We report the complete enumeration of automorphic equitable partitions and partial results for the general case.

Sections 8–10 reflect our efforts to understand a few of the computer aided results and to present them in a computer free form (completely, or at least relatively).

In Section 8 we deal with the graph $NL_2(10)$ and describe it in a sense locally, that is via so-called metric equitable partitions, obtained from the embeddings of a quadrangle and a graph $K_{3,3}$ without one edge (Atkinson configuration), with 4 and 8 cells respectively. The spectrum of the second partition contains all of the eigenvalues of $NL_2(10)$ (the case of full spectrum). Some modifications of the famous Robertson model (see [21]) appear in Section 9, while in Section 10 we present a handful of models for some other graphs in $\mathfrak{F}$.

It is worthy to mention that for generations of mathematicians, the graphs in $\mathfrak{F}$ appear as a source of ongoing aesthetic admiration; see, especially the home page of Andries Brouwer [6]. We believe that some of our models shed a new light on those graphs.

Last but not least, is that as a rule, the models we suggest rely on a relatively small subgroup of the group $Aut(\Gamma)$, for $\Gamma \in \mathfrak{F}$. In this way we provide promising patterns for those researchers who wish in future to face a challenge: to try to construct new primitive triangle free strongly regular graphs, cf. Section 11.

2 Strongly regular graphs: a brief survey

An undirected graph Γ is called a *strongly regular graph* (SRG) with parameters (v, k, λ, μ) if it is a regular graph of order v and valency k, and every pair of adjacent vertices has exactly λ common neighbors, while every pair of non-adjacent vertices has exactly μ common neighbors. Sometimes we use an extended set of parameters, (v, k, l, λ, μ), where l is the number of non-neighbors of a vertex, that is $l = v - k - 1$.

If $A = A(\Gamma)$ is the adjacency matrix of an undirected graph Γ, then Γ is strongly regular if and only if $A^2 = kI + \lambda A + \mu(J - I - A)$. This implies that $(I, A, J - I - A)$ is the standard basis of a rank 3 homogeneous coherent algebra. In other words, $(\Delta, \Gamma, \overline{\Gamma})$ are the basic graphs of a rank 3 symmetric association scheme (here $\overline{\Gamma}$ is the complement of Γ, while Δ contains all the loops). The adjacency matrix of a strongly regular graph has exactly 3 distinct eigenvalues. For a strongly regular graph we denote by:

- $r > s$, the two eigenvalues of $A(\Gamma)$ different from k; r is always positive, while s is always negative;

- f, g, the multiplicity of the eigenvalues r, s respectively.

A formula for f and g is given by $g, f = \frac{1}{2}\left[(v-1) \pm \frac{2k+(v-1)(\lambda-\mu)}{\sqrt{(\lambda-\mu)^2+4(k-\mu)}}\right]$.

A quadruple of parameters (v, k, λ, μ) for which f and g as given by the preceding formulas are positive integers is called a feasible set of parameters. See [6] for a list of feasible parameter sets with some information about known graphs for some of the sets.

A strongly regular graph Γ is called *primitive* if both Γ and its complement $\overline{\Gamma}$ are connected. This is equivalent to primitivity of the related association scheme $(\Delta, \Gamma, \overline{\Gamma})$.

3 Computer algebra tools in AGT

During the last two decades the significance of the use of computer algebra systems in AGT increased drastically. Such systems are exploited in the search for new combinatorial objects, the enumeration of objects with prescribed parameters and the understanding of algebraic and structural properties of a given combinatorial object Γ (such as the automorphism group $Aut(\Gamma)$, its action and orbits on the ingredients of Γ, enumeration of substructures of Γ, embeddings of Γ into larger structures, etc.)

An ideal case is when the computer aided understanding of an object Γ is followed by further theoretical generalizations. The foremost goal of this paper is to share with the reader many new interesting properties of the graphs in the family $\mathfrak{F}$ in order to promote the search for new primitive tfSRGs.

Below are the main tools we use.

- **COCO** is a set of programs for dealing with coherent configurations, including construction, enumeration of subschemes, and calculation of automorphism groups.

  Developed in 1990-2, Moscow, USSR, mainly by Faradžev and Klin [8], [9].

- **WL-stabilization** – Weisfeiler-Leman stabilization is an efficient algorithm for calculating coherent closure of a given matrix (see [25], [3]). Two implementations of the WL-stabilization are available (see [2]).

- **GAP** an acronym for "Groups, Algorithms and Programming", is a system for computation in discrete abstract algebra [10], [22]. It supports easy addition of extensions (packages, in gap nomenclature), that are written in the GAP programming language which can add new features to the GAP system.

  One such package, GRAPE [23], is designed for construction and analysis of finite graphs. GRAPE itself is dependent on an external program, nauty [18] in order to calculate the automorphism group of a graph. Another package is DESIGN, used for construction and examination of block designs.

- **COCO v.2** – The COCO v.2 initiative aims to re-implement the algorithms in COCO, WL-stabilization and DISCRETA as a GAP package. In addition, the new package should essentially extend abilities of current version basing on new theoretical results obtained since the original COCO package was written.

We refer to [14] for a more detailed discussion of the ways in which computer algebra systems are used in AGT for the purpose of experimentation and further theoretical generalizations.

4 The seven known primitive triangle-free strongly regular graphs

A graph Γ is called triangle free if it admits no triangles, that is cliques of size 3. If Γ is also a strongly regular graph then it is called a *triangle free strongly regular graph* (tfSRG for short). A graph is triangle free if any two neighbors have no common neighbors, therefore a tfSRG is an SRG with $\lambda = 0$.

The 7 known primitive tfSRGs, with orders from 5 to 100 vertices are:

1. Pentagon with parameters $(5, 2, 0, 1)$. Its automorphism group is D_5 of order 10.

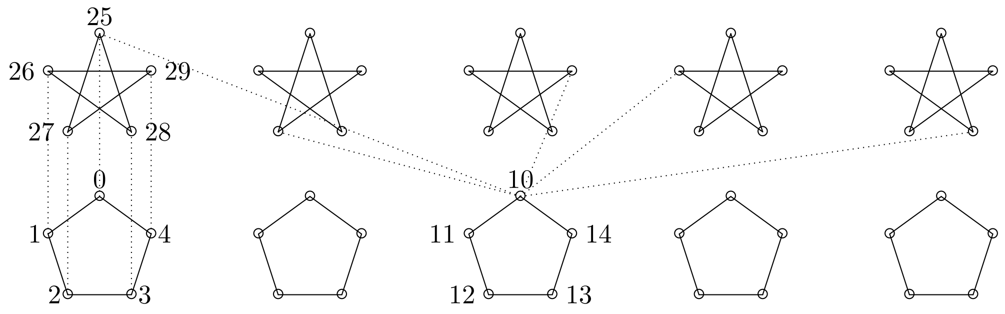


Figure 1: Hoffman-Singleton graph, Robertson model

2. Petersen graph with parameters $(10, 3, 0, 1)$. Its automorphism group is isomorphic to S_5 of order 120. A simple model has as vertices 2-subsets of a set of size 5, with two vertices adjacent if the subsets are disjoint.

3. Clebsch graph with parameters $(16, 5, 0, 2)$. Usually denoted by $\square_5$. Its automorphism group is isomorphic to $(S_5 \wr S_2)^{pos}$ of order 1920. A simple model is 4-dimensional cube Q_4 together with long diagonals, or Cayley graph: $CAY(E_{2^4}, \{0001, 0010, 0100, 1000, 1111\})$.

4. Hoffman-Singleton graph (HoSi) with parameters $(50, 7, 0, 1)$. Its automorphism group is isomorphic to $P\Sigma U(3, 5^2)$ of order 252000 ([4]). The simplest model is Robertson model ([21]): 5 pentagons marked $P_0, \ldots, P_4$ and 5 pentagrams marked $Q_0, \ldots, Q_4$ with vertex i of P_j joined to vertex $i + jk \pmod 5$ of Q_k.

5. Gewirtz (or Sims-Gewirtz) graph with parameters $(56, 10, 0, 2)$. Its automorphism group of order 80640 is a non split extension of $PSL_3(4)$ by E_{2^2}. A simple model is as the induced subgraph of $NL_2(10)$ on the common non-neighbors of two adjacent vertices.

6. Mesner graph with parameters $(77, 16, 0, 4)$. The automorphism group is of order 887040 and is isomorphic to the stabilizer of a point in the automorphism group of $NL_2(10)$. One simple model is: induced subgraph of $NL_2(10)$ on non-neighbors of a vertex.

7. $NL_2(10)$ with parameters $(100, 22, 0, 6)$, also known as the Higman-Sims graph. Its automorphism group contains Higman-Sims sporadic simple group as a subgroup of index 2. We refer to [15] for a detailed presentation of the original construction of this graph by Dale Mesner, as it appeared on [19], [20].

Recall that the graph $NL_2(10)$ on 100 vertices was constructed twice: by Dale Mesner in his Thesis in 1956 [19] and later on by Higman and Sims in 1968, when they discovered a new sporadic simple group. Moreover, in 1964 Mesner proved its uniqueness. We denote this graph by $NL_2(10)$, following the parametrization for a family of putative graphs, introduced by Mesner in [20]. In [15] the history of this discovery, as well as of related events is discussed with many details. A new wave of interest in tfSRGs, stimulated by [15], was especially productive for us. It became clear that this is the time to arrange a serious comprehensive investigation of diverse structural properties of the graphs in the family $\mathfrak{F}$. The goal is to attract the attention of researchers to such an information in order to provide new ideas how efforts for the search of new tfSRGs should be arranged. We also wish to share with the reader some surprising new features of the graphs in the family $\mathfrak{F}$, observed with the aid of a computer.

5 Embeddings of tfSRGs into known primitive tfSRGs

Usually in graph theory a graph $\Delta = (V', E')$ is called a *subgraph* of a graph $\Gamma = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. Of special interest are *induced* subgraphs, in this case $E' = E \cap \left\{ {V' \atop 2} \right\}$, where $\left\{ {X \atop 2} \right\}$ is the set of all 2-subsets of X.

Proposition 1. *A subgraph Δ of diameter 2 of a graph Γ with no triangles is an induced subgraph.*

Proof. Assume that Δ is not induced, then there exists an edge $\{u, w\} \in E$, $\{u, w\} \notin E'$ with $u, w \in V'$. Since the diameter of Δ is 2, there exist a v which is adjacent in Δ (and therefore in Γ) to u, w, thus $\{u, v, w\}$ is a triangle in Γ, which is a contradiction. □

The enumeration of all tfSRGs inside of known tfSRGs was the first serious computational challenge in this project. The results of its complete solution are discussed below.

A non-empty imprimitive tfSRG is either a complete bipartite graph ($K_{l,l} \cong \overline{2 \circ K_l}$), or a graph $s \circ K_2$ consisting of s edges with no common vertices. Note that $K_{2,2}$ is simply a quadrangle.

Lemma 2. *Graph $NL_2(10)$ does not contain subgraph $K_{3,3}$.*

Proof. Originally, we obtained this result with the aid of a computer. In fact, easy proof follows from the non-edge decomposition of $\Gamma = NL_2(10)$ (see e.g. [15]), without any assumptions about its uniqueness. Indeed, let Δ be a subgraph of Γ, which is isomorphic to $K_{3,3}$. Assume that two bipartite parts of Δ consist of subsets $\{a, b, c\}$, $\{d, e, f\}$ of vertices. Consider now non-edge decomposition of Γ with respect to $\{a, b\}$, then c belongs to the cell of size 60, while d, e, f belong to the cell of size 6, thus c has just $g = 2$ neighbors in the latter cell, a contradiction with the valency of c, equal to 3 in Δ. □

As a corollary, we obtain that every known primitive tfSRG does not contain $K_{3,3}$, and moreover does not contain $K_{l,l}$, when $l \geq 3$.

Another simple theoretical result (see e.g. [28]) claims that the number of quadrangles in an SRG with parameters (v, k, l, λ, μ) is equal to $\frac{\frac{vk}{2}\binom{\lambda}{2} + \frac{vl}{2}\binom{\mu}{2}}{2}$. When $\lambda = 0$ this reduces to $\frac{vl\mu(\mu-1)}{8}$. Similar formula may be obtained for the number of $2 \circ K_2$. The numbers of imprimitive tfSRGs inside of graphs in $\mathfrak{F}$ are given in Table 1 in the supplement, while the number of orbits of such embeddings (with respect to $Aut(\Gamma)$ for $\Gamma \in \mathfrak{F}$) are given in Table 2.

Note that there is no induced subgraph of $NL_2(10)$, isomorphic to $12 \circ K_2$, so both tables in Supplement A end with $s = 11$ for $s \circ K_2$.

Similar computer results are presented in Tables 3 and 4 (Supplement A) for the embeddings of a graph from the family $\mathfrak{F}$ into a larger graph in $\mathfrak{F}$.

Here the final picture is less sophisticated. There are just two cases where for pair of graphs $X, Y \in \mathfrak{F}$ there exist more than one orbit of embeddings (up to action of $Aut(Y)$) of X into Y. Namely, in these terms, there are 9 embeddings of Petersen graph into Mesner graph and 5 embeddings into $NL_2(10)$. Computer free understanding of these embeddings remains a quite interesting task for future investigations of the links between the graphs in the family $\mathfrak{F}$.

In all other cases there exists (up to isomorphism) at most one embedding for considered pair (X, Y). For most pairs there is a reasonably clear explanation in literature, such as two HoSi inside $NL_2(10)$, see e.g. [12], as well as Robertson model of HoSi [21], which explains positions in it for both pentagon and Petersen graph.

Beside this there are two more embeddings, which are not immediately evident.

The first pair is the embedding of the Petersen graph into Gewirtz graph. In fact, in this embedding the Petersen graph appears as a subgraph of the second constituent of the Gewirtz graph, which in turn is a graph Δ of valency 8 on 45 vertices. It turns out that $Aut(\Delta) \cong Aut(S_6)$. This graph Δ generates a non-Schurian association scheme with three classes, see e.g. [24]; [13], example W_{38}. The graph Δ may be also described as the distance 2 graph of the distance transitive graph of diameter 4 and valency 4 on 45 vertices, a.k.a generalized octagon $GO(2,1)$, a.k.a line graph of Tutte's 8-cage (see [5] for more details). In this context, the task of explanation of the embedding of Petersen graph into Δ seems to be a nice exercise in AGT, though out of the framework of this presentation.

The second exceptional pair is the embedding of the Clebsch graph $\square_5$ into $NL_2(10)$. This pair is of a definite independent interest and thus is considered separately in Section 10.

Finally we mention that the classification of cocliques in graphs from $\mathfrak{F}$ may be regarded as a degenerate case of subject in this section. Indeed, empty graph is a particular case of an imprimitive tfSRG. We however disregard this problem, referring to information presented on the home page of Andries Brouwer [6].

6 Equitable partitions

Let $\Gamma = (V, E)$ be a graph. A partition τ of the vertex set V, $\tau = \{V_1, \ldots, V_s\}$ is called *equitable partition* (briefly, EP) if for $i, j \in \{1, \ldots, s\}$, the numbers $|\Gamma(v) \cap V_j|$ are equal for all $v \in V_i$. Here $\Gamma(v) = \{u \in V | \{u, v\} \in E\}$. Usually an EP τ is accompanied by *intersection diagram*, which represents a kind of quotient graph, Γ/τ, on which all intersection numbers are depicted. Many such diagrams appear in [5]. The quotient graph Γ/τ is, in fact, a multigraph. Its (collapsed) adjacency matrix B consists of all intersection numbers.

Obviously, entries of B are non-negative integers, and for a regular graph Γ of valency k the sum of each row in B is k.

If H is a subgroup of $Aut(\Gamma)$, then the set of orbits of H is an EP of Γ. Such an EP is called *automorphic* (briefly AEP).

Proposition 3 ([11])**.** *Let Γ be a graph, $A = A(\Gamma)$ the adjacency matrix of Γ. If a partition τ is EP of Γ with matrix B then the characteristic polynomial of B divides the characteristic polynomial of A.*

In fact, there are more necessary conditions for a prescribed matrix B to be the adjacency matrix of a suitable EP of Γ, which are formulated in terms of spectral graph theory, see e.g. [7]. They create a solid background for a clever search of potential EPs in a given graph Γ.

7 Search for equitable partitions

In this project we distinguished two alternative problems for the complete search of EPs in graphs of $\mathfrak{F}$.

First problem is to enumerate all automorphic EPs. Here we strongly rely on the level of group theoretical "intellect" of GAP. Indeed, for groups of relatively small order (say up to a few thousands) GAP allows to establish efficiently the structure of a considered group, as well as the lattice of its subgroups.

However, for four graphs in $\mathfrak{F}$ the group $G = Aut(\Gamma)$ has larger order, thus with the growing of $|G|$ extra theoretical input is required. For example, in case of $G = Aut(NL_2(10))$ we were using information from the website of the Atlas of Finite Group Representations ([26]) about the maximal subgroups of G (this information goes back to [17]). This knowledge, together with ad hoc tricks inside of GAP made it possible to describe all subgroups of G up to equivalency classes with respect to EPs. The fact that all groups $G = Aut(\Gamma)$, $\Gamma \in \mathfrak{F}$, are subgroups of $Aut(NL_2(10))$ was also quite beneficial.

Finally, we successfully enumerated all automorphic EPs for all $\Gamma \in \mathfrak{F}$.

The second problem is much more sophisticated: to enumerate all EPs for all graphs in $\mathfrak{F}$.

For the smallest three graphs in $\mathfrak{F}$ the results were achieved via simple brute force, see corresponding tables in Supplement B.

For HoSi we used a few complementary strategies in the search for all EPs. First, we attacked partitions with "large" number of cells, say $s > 5$. Here we introduced extra parameter: the size of the smallest cell, which varies from 1 to 9. Each such case was considered separately.

On the second stage, we step by step enumerated cases $2 \leq s \leq 5$, by enumerating possible collapsed adjacency matrices for each case, and for any such matrix, enumerating all EPs.

The description of our computer activities together with detailed discussion of the results is presented in [28]. An extra advantage of the enumeration in HoSi is that it serves as a kind of a pleasant interaction of a human and a computer. Indeed, in spite of the fact that the main part of computation was fulfilled by GAP, a human can follow search, explain its logic and step by step to comprehend all ongoing results.

We admit however that the extra advantage of HoSi, is that here parameter μ takes its smallest possible value of 1 (for a primitive SRG). As soon as $\mu > 1$ and v is growing, the problem is becoming essentially more difficult.

This is why already for the Gewirtz graph, with $v = 56$ we only succeeded in enumeration of all EPs for which the stabilizer has order at least 2. There are 688 such EPs. The enumeration of the rigid EPs cannot be completed by a computer in a reasonable time.

The full enumeration of all EPs for the largest two graphs in $\mathfrak{F}$ (on 77 and 100 vertices) at this stage looks intractable. Only modest particular cases may be proceeded efficiently.

We refer to Supplement B for Tables 5, 6 and 7, which summarize our activities for both considered problems, see [28] for more information.

As was mentioned in the introduction, one of the main goals of this project is to detect a number of "nice" EPs, which may serve as a training model for future search of new primitive tfSRGs. At this stage this kind of job only partially relies on the support from computer, finally a human's insight still turns out to be very significant. Below we report about interesting interpretations created by us for a handful of detected EPs.

8 Quadrangle and Atkinson EPs of $NL_2(10)$

Recall that $\Gamma = NL_2(10)$ has $\frac{100\cdot 77\cdot 6\cdot 5}{8} = 28875$ quadrangles. This is a simple result, obtained on a combinatorial level without any knowledge of structure of Γ.

Provided that Γ and $G = Aut(\Gamma)$ are known, we easily obtain (with the use of a computer) that all quadrangles in Γ form one orbit under G and stabilizer H of a quadrangle in G has order 3072, is isomorphic as abstract group to $(\mathbb{Z}_4 \times \mathbb{Z}_2).(\mathbb{Z}_4 \times \mathbb{Z}_2)) : \mathbb{Z}_2) : \mathbb{Z}_2) : \mathbb{Z}_3) : \mathbb{Z}_2) : \mathbb{Z}_2$ and has in action on $V(\Gamma)$ four orbits of lengths 4, 8, 24 and 64.

Thus we may consider an automorphic EP of size $s = 4$. There is sense to try again combinatorial arguments in order to determine all invariants of this EP without any prior knowledge of Γ and its group. Fortunately, this turns out to be possible.

Proposition 4. *Let τ be a metric decomposition with respect to a prescribed quadrangle Q inside of Γ. Then τ is EP, $s = 4$, sizes of cells are 4, 8, 24, 64 and (with respect to this* *ordering of cells),* $B = \begin{pmatrix} 2 & 4 & 0 & 16 \\ 2 & 0 & 12 & 8 \\ 0 & 4 & 2 & 16 \\ 1 & 1 & 6 & 14 \end{pmatrix}$.

See proof in [28].

In principle, next stage should be to describe a model of $NL_2(10)$ in terms of the group H of order 3072 or one of its subgroups which is transitive on all cells of the quadrangle EP. Nevertheless, in this text we are avoiding this task. Instead of, we prefer to split the quadrangle EP to one with larger number of cells and to proceed with that new one.

A note [1] was published before the announcement of CFSG, The author considered 13 parameter sets for tfSRGs, which may be also rank 3 graphs, with the number v of vertices where $100 < v \leq 1000$ and proved that such graphs can not be constructed with the aid of known 2-transitive permutation groups. Part of exposed details is related to consideration of a configuration $\mathfrak{A}$ (graph $K_{3,3}$ minus one edge) in putative graph.

We got impression that [1] is overseen through a generation of mathematicians and it might be helpful to analyze existence of $\mathfrak{A}$ (we call it Atkinson configuration) in known tfSRGs.

It turns out that $\mathfrak{A}$ appears only in graphs on 77 and 100 vertices. Here we analyze its position in $\Gamma = NL_2(10)$.

Let us fix at $\mathfrak{A}$ a quadrangle, for example with the vertices $\{a, b, c, f\}$. Then two extra vertices have two neighbors in the selected quadrangle, namely ends of two non-edges respectively.

Now we embed $\mathfrak{A}$ into the considered above quadrangle decomposition splitting in it cell of size 8 into two cells of size 2 (exactly our $\{d, e\}$) and the remainder. This immediately implies split of the cell of size 64 into two cells of sizes 16 and 48. The cell of size 24 is forced to be split into subcells of sizes 12, 6, 6, depending on the number of neighbors in $\{d, e\}$ being 1, 2 and 0, respectively.

A more careful analysis (see [28]) shows that we are still getting an EP.

Proposition 5. *A metrical decomposition $\tau(\mathfrak{A})$ of a tfSRG with the parameters of $NL_2(10)$ with respect to configuration $\mathfrak{A}$ is EP with $s = 8$ and cells of sizes 4, 2, 16, 48, 6, 12, 6, 6. It has collapsed matrix B as follows, with $Spec(B) = \{22, 2^5, (-8)^2\}$.*

$$B = \begin{pmatrix} 2 & 1 & 4 & 12 & 3 & 0 & 0 & 0 \\ 2 & 0 & 8 & 0 & 0 & 6 & 6 & 0 \\ 1 & 1 & 2 & 12 & 0 & 3 & 0 & 3 \\ 1 & 0 & 4 & 10 & 1 & 3 & 2 & 1 \\ 2 & 0 & 0 & 8 & 0 & 6 & 2 & 4 \\ 0 & 1 & 4 & 12 & 3 & 2 & 0 & 0 \\ 0 & 2 & 0 & 16 & 2 & 0 & 0 & 2 \\ 0 & 0 & 8 & 8 & 4 & 0 & 2 & 0 \end{pmatrix}$$

Originally, the proof was obtained with the aid of computer analysis of $\Gamma = NL_2(10)$. Later on an outline of a similar proof as for quadrangle configuration was again obtained without prior knowledge of Γ.

Further consideration of $\tau(\mathfrak{A})$ was fulfilled inside of known graph Γ and its group $Aut(\Gamma)$. It turns out that $H = Aut(\tau(\mathfrak{A})) \cong D_4 \times S_4$ is a group of order 192. H has exactly 8 orbits on the set $V(\Gamma)$, thus $\mathfrak{A}$ is an automorphic EP.

Describing further stabilizers of H on each of the 8 orbits we elaborated the following ad hoc model of Γ.

We start from the auxiliary graph Δ depicted in Figure 2.

We identify elements of each orbit in $\tau(\mathfrak{A})$ with suitable sets of structures, described with the aid of Δ. In fact, each set of structures appears as orbit of action of the group H. Thus it is enough to describe a representative of each orbit.

For example, the elements in the cell of size 16 are the vertices of Δ, while the elements of the cell of size 4 are the four columns of Δ. The most tricky to find was a representation of the 7th cell, of size 6.

We think that this piecewise model of Γ, appearing from a reasonably small solvable group of order 192 may serve as an optimistic message for those researchers who are hunting for new tfSRGs, see also comments in Section 11.

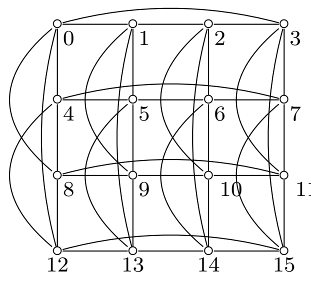


Figure 2: Auxiliary graph Δ

9 Some modifications of Robertson model

The Robertson model of the Hoffman-Singleton graph (see Figure 1) partitions the vertices into five pentagons and five pentagrams, and describes a simple arithmetic rule for the edges connecting pentagons and pentagrams (see Section 4). Let p_{ij} (q_{ij}) be vertex j of pentagon (pentagram) i.

Since $NL_2(10)$ has an EP to two cells of size 50, where the induced graph on each cell is isomorphic to HoSi, this model can be extended to a model of $NL_2(10)$.

In both cases we get an EP with a collapsed adjacency matrix that has a transitive automorphism group (a permutation $g \in S_n$ is an automorphism of a matrix $M \in \mathbb{F}^{n\times n}$ if the permutation matrix M_g commutes with M). We are therefore interested in equitable partitions with this property.

In the list of equitable partitions of HoSi we find another such partition with 10 cells of size five. The collapsed adjacency matrix of this partition is M. The spectrum of M is $\{7, 2^4, (-3)^5\}$, so it has the full spectrum of HoSi.

The cells of this partition can be described in the Robertson model of HoSi: for each $i \in [1, 5]$, we construct two cells: $\{p_{1,1+i}, p_{2,1+i}, p_{3,2+i}, p_{4,4+i}, p_{5,2+i}\}$, $\{q_{1,1+i}, q_{2,1+i}, q_{3,5+i}, q_{4,3+i}, q_{5,5+i}\}$.

$$M = \begin{pmatrix} 0&2&2&0&0&0&0&1&1&1\\ 2&0&1&0&1&2&0&0&1&0\\ 2&1&0&2&0&0&1&0&0&1\\ 0&0&2&0&1&1&2&0&0&1\\ 0&1&0&1&0&2&0&1&0&2\\ 0&2&0&1&2&0&1&0&1&0\\ 0&0&1&2&0&1&0&1&2&0\\ 1&0&0&0&1&0&1&0&2&2\\ 1&1&0&0&0&1&2&2&0&0\\ 1&0&1&1&2&0&0&2&0&0 \end{pmatrix}$$

If we consider this matrix M as the adjacency matrix of a color graph, then the basic graph for the color 1 is the Petersen graph, while the basic graph for the color 2 is a cycle of length 10. For comparison, acting similarly with the Robertson partition we get a graph $K_{5,5}$.

The stabilizer of the (ordered) partition in $Aut(\mathrm{H}oSi)$ is isomorphic to the dihedral group D_5 of order 10, as is the automorphism group of M.

The coherent closure of the partition (more specifically, of the graph with its vertices colored according to the partition) is a Schurian coherent configuration of rank 300, with the stabilizer of the partition as its automorphism group.

This EP of HoSi can be extended to an EP of $NL_2(10)$, of 20 independent sets of size 5. In this case, the matrix has spectrum $\{22, 2^{13}, (-8)^6\}$, so this EP has full spectrum as well.

The automorphism group of this matrix has order 240 and is isomorphic to $\mathbb{Z}_2 \times S_5$, while the stabilizer of the ordered partition is again D_5 of order 10.

The coherent closure of the partition is a Schurian coherent configuration of rank 1200, with the stabilizer of the partition as its automorphism group.

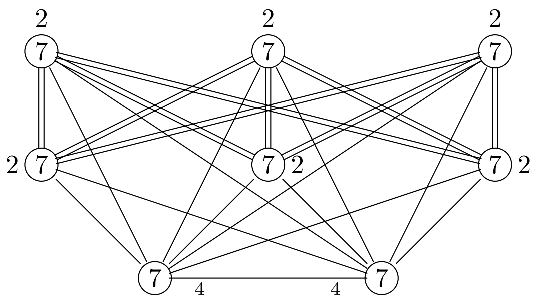


Figure 3: AEP of Gewirtz graph by a semiregular subgroup of order 7

10 A few more models

The calculated data, both about embeddings of tfSRGs inside tfSRGs and about EPs can be used to present new models and constructions for tfSRGs. Here we offer a few examples, without going into a lot of details:

Example 1 (Two Wagner graphs in Clebsch graph)**.** *Wagner graph, also known as Möbius ladder of order 8, denoted by M_8 is a cubic graph on 8 vertices. A simple model for it is a $Cay(\mathbb{Z}_8, \{1, -1, 4\})$. The spectrum is $\{3, 1^2, -1\}$ and $Aut(M_8) = D_8$.*

In one of the three EPs of $\Box_5$ into two cells of size 8, the induced subgraph on each cell is the Wagner graph. The stabilizer of ordered partition inside $Aut(\Box_5)$ is D_8.

Example 2 (Non-automorphic EP of HoSi)**.** *We can construct an EP of HoSi into two cells by taking one of the cells to be the 20 vertices of two Petersen graphs (any two pentagons and two pentagrams in Robertson model). The stabilizer of this partition is D_{10} of order 20, so it is not automorphic.*

Example 3 (Inside Gewirtz graph: six 7-cycles)**.** *There is one automorphic partition of Gewirtz graph into 8 cells of size 7. The induced graph on 6 of the cells is a cycle of length 7, while the induced graph on the other two cells has no edges.*

From the intersection diagram (Figure 3) we see that this partition can be merged to an equitable partition with 2 cells of sizes 42 and 14 with adjacency matrix $\begin{pmatrix} 8 & 2 \\ 6 & 4 \end{pmatrix}$.

The induced graph on the cell of size 42 of valency 8 is a coherent graph. Its coherent closure is a rank 6 imprimitive non-Schurian association scheme. This graph is not a rational graph, with spectrum $\{8, 2^{22}, (-1)^7, (1+\sqrt{2})^6, (1-\sqrt{2})^6\}$.

The induced graph on the cell of size 14 is a bipartite graph with two parts of size 7. It is the co-Heawood graph, that is the bipartite complement to the Heawood graph, the Levi graph of Fano plane. This is by necessity true, since a regular bipartite graph of valency 4 and order 14 in which every vertex belongs to exactly one quadrangle is the co-Heawood graph.

A. Brouwer ([6]) notes that this co-Heawood graph is a subgraph of the Gewirtz graph.

Example 4 (Gewirtz graph inside of Mesner graph through the group $\mathbb{Z}_7$)**.** *We start with a semiregular subgroup $\mathbb{Z}_7$ of order 7, this time inside of the automorphism group of Mesner graph. We get an AEP with 11 cells of size 7; the collapsed matrix N of order 11 is presented in [28]. It turns out that $Aut(N) \cong S_2 \wr S_3$ (the wreath product) is a group of order 72 with orbits of sizes $6, 2, 1^3$. There are three non-equivalent possibilities to merge from this AEP a* *non-automorphic EP with 3 cells of sizes 14, 21, 42 and a collapsed matrix* $\begin{pmatrix} 4 & 6 & 6 \\ 4 & 0 & 12 \\ 2 & 6 & 8 \end{pmatrix}$*, which has full spectrum.*

Each of the cases provides also an EP for the Gewirtz graph with two cells of sizes 14 and 42. Thus, as a by product, we obtain other (cf. Example 3) embeddings of the co-Heawood graph into the Gewirtz graph. This example demonstrates "irregularities", which may cause interesting EPs, unpredictable from the first sight.

Example 5 (Clebsch graph inside $NL_2(10)$)**.** *Recall that up to the automorphism group of* $\Gamma = NL_2(10)$*, there is one embedding of* $\square_5$ *into* Γ*. The equitable partition resulting from this embedding allows us to describe another model for* $NL_2(10)$*.*

We start from a group $H_2 = \mathbb{Z}_4 \times S_4$ *of order 96, with orbits of lengths 16, 16, 48, 12, 8. The collapsed adjacency matrix of the corresponding automorphic equitable partition is* B''*, (shown below). Clearly, the induced graph on the first cell of the partition is isomorphic to* $\square_5$*.*

In principle, one may try as in previous sections to reconstruct the corresponding matrix B *from scratch, not relying on knowledge of the graph* Γ *and its group. A promising starting point would be justification of existence of a coclique of size 8 and its adjacency with the induced* $\square_5$*.*

We, however, prefer to exploit at this stage another possibility. Namely, it turns out that the stabilizers of $\mathfrak{A}$ *and* $\square_5$ *inside of* Γ *have maximal possible intersection* $K = \mathbb{Z}_4 \times S_3$ *of order 24.* K *defines an AEP with 11 cells of sizes 2, 4, 4, 6, 6, 6, 12, 12, 12, 12, 24 and matrix* B'*.*

$$B' = \begin{pmatrix} 0 & 2 & 2 & 0 & 0 & 6 & 0 & 6 & 0 & 6 & 0 \\ 1 & 2 & 1 & 3 & 0 & 0 & 3 & 3 & 3 & 0 & 6 \\ 1 & 1 & 2 & 0 & 3 & 0 & 3 & 0 & 3 & 3 & 6 \\ 0 & 2 & 0 & 0 & 4 & 2 & 2 & 0 & 2 & 6 & 4 \\ 0 & 0 & 2 & 4 & 0 & 2 & 2 & 6 & 2 & 0 & 4 \\ 2 & 0 & 0 & 2 & 2 & 0 & 4 & 0 & 4 & 0 & 8 \\ 0 & 1 & 1 & 1 & 1 & 2 & 4 & 3 & 2 & 3 & 4 \\ 1 & 1 & 0 & 0 & 3 & 0 & 3 & 2 & 3 & 3 & 6 \\ 0 & 1 & 1 & 1 & 1 & 2 & 2 & 3 & 4 & 3 & 4 \\ 1 & 0 & 1 & 3 & 0 & 0 & 3 & 3 & 3 & 2 & 6 \\ 0 & 1 & 1 & 1 & 1 & 2 & 2 & 3 & 2 & 3 & 6 \end{pmatrix} \qquad B'' = \begin{pmatrix} 5 & 4 & 9 & 3 & 1 \\ 4 & 2 & 12 & 0 & 4 \\ 3 & 4 & 11 & 3 & 1 \\ 4 & 0 & 12 & 2 & 4 \\ 2 & 8 & 6 & 6 & 0 \end{pmatrix}$$

It is easy to recognize how B' *is split from the matrix* B *of the metric EP of the configuration* $\mathfrak{A}$*. Then we merge the* B' *into* B''*:*

The cell of size 48 is a merging of the last 3 cells, the cell of size 8 is merging of cell of sizes 2 and 6 (5th cell), while the cell of size 12 is a merging of two remaining cells of size 6. Each cell of size 16 is a merging of cells of sizes 4 and 12.

Example 6 (Reconstruction of $NL_2(10)$ from a subgroup of order 2)**.** *Using information in the atlas [26] (or direct calculation using GAP), we see that* $G = Aut(NL_2(10))$ *contains one conjugacy class of involutions with no fixed points. From such an involution,* i*, we get an AEP into 50 cells of size 2. We investigated the quotient graph* $\widetilde{\Gamma}$ *of this AEP, and its collapsed adjacency matrix, denoted by* B*. The structure of* $\widetilde{\Gamma}$ *may be recovered by considering the centralizer* $C_G(i)$*. However, we prefer to rely on information of combinatorial nature. The graph* $\widetilde{\Gamma}$ *has a natural EP into cells of sizes 20 and 30 (representing edges and non-edges in* $NL_2(10)$*).*

The induced (color) graph on 20 vertices is the graph $K_{10,10}$ *from which a 1-factor is removed.*

The induced graph on 30 vertices is the incidence graph of generalized quadrangle of order 2, $GQ(2)$*.*

This equitable partition corresponds to an equitable partition of $\Gamma = NL_2(10)$*, into two cells of sizes 40 and 60.*

11 Concluding comments

The presentation in this extended abstract is a tip of iceberg. Much more concrete information may be found in [28], while [27], subject of ongoing polishing and development, will reflect all detected computer data about the graphs from family $\mathfrak{F}$. We see also more potential to return again to construction of further models for graphs in $\mathfrak{F}$, in particular relying on non-automorphic EPs. The new attempts to achieve, at least in part, enumeration of all EPs of restricted size of cells for graphs on 77 and 100 vertices are also on agenda. The paper [16] as well as a number of private communications of M. Mačaj to author MK, create a background to attack in nearest future the problem of existence of new tfSRGs with relatively small parameters. The parameter set on 162 vertices seems to be first interesting and realistic target.

It should be mentioned that the results in [16], as well as in numerous private communications of M. Mačaj to the author MK, show that a putative new primitive triangle free strongly regular graph Γ with prescribed parameters will have a relatively small automorphism group $Aut(\Gamma)$; very concrete restrictions on the cycle structure of elements in $Aut(\Gamma)$ are typically available. Transformation of those restrictions to the language of equitable partitions may be regarded as a reasonable starting platform for attempts to construct a new tfSRG.

Note, however, that many experts in AGT do not believe at all that a new tfSRG exists. Needless to say that this question creates a great challenge for the modern scientific community.

Acknowledgments

We thank Andy Woldar for ongoing collaboration in this project. We acknowledge Sven Reichard and Christian Pech for sharing with us their draft versions of COCO II. Ongoing communication with Martin Mačaj is very helpful and stimulative. We also are pleased to thank Peter Cameron and Jozef Širáň for interest expressed to the topic of our research.

References

[1] M. D. Atkinson. On rank 3 groups having $\lambda = 0$. *Canad. J. Math.*, 29(4):845–847, 1977.

[2] L. Babel, S. Baumann, and M. Luedecke. Stabcol: An efficient implementation of the weisfeiler-leman algorithm. Technical Report TUM-M9611, Technical University Munich, 1996.

[3] L. Babel, I. V. Chuvaeva, M. Klin, and D. V. Pasechnik. Algebraic combinatorics in mathematical chemistry. methods and algorithms. ii. program implementation of the weisfeiler-leman algorithm. Technical Report TUM-M9701, Fakultät für Mathematik, TU Münc, 1997. `http://www-lit.ma.tum.de/veroeff/html/960.68019.html`.

[4] C. T. Benson and N. E. Losey. On a graph of Hoffman and Singleton. *J. Combinatorial Theory Ser. B*, 11:67–79, 1971.

[5] A. E. Brouwer, A. M. Cohen, and A. Neumaier. *Distance-regular graphs*, volume 18 of *Ergebnisse der Mathematik und ihrer Grenzgebiete (3) [Results in Mathematics and Related Areas (3)]*. Springer-Verlag, Berlin, 1989.

[6] A.E. Brouwer. `http://www.win.tue.nl/~aeb/graphs/`.

[7] Dragoš Cvetković, Peter Rowlinson, and Slobodan Simić. *An introduction to the theory of graph spectra*, volume 75 of *London Mathematical Society Student Texts*. Cambridge University Press, Cambridge, 2010.

[8] I. A. Faradžev and M. H. Klin. Computer package for computations with coherent configurations. In *Proc. ISSAC*, pages 219–223, Bonn, 1991. ACM Press.

[9] I. A. Faradžev, M. H. Klin, and M. E. Muzichuk. Cellular rings and groups of automorphisms of graphs. In *Investigations in algebraic theory of combinatorial objects*, volume 84 of *Math. Appl. (Soviet Ser.)*, pages 1–152. Kluwer Acad. Publ., Dordrecht, 1994.

[10] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.4.12*, 2008.

[11] Chris Godsil and Gordon Royle. *Algebraic graph theory*, volume 207 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 2001.

[12] Paul R. Hafner. On the graphs of Hoffman-Singleton and Higman-Sims. *Electron. J. Combin.*, 11(1):Research Paper 77, 33 pp. (electronic), 2004.

[13] M. Klin, C. Pech, and P.-H. Zieschang. Flag algebras of block designs. i. initial notions. steiner 2-designs and generalized quadrangles, November 1998.

[14] Mikhail Klin, Christian Pech, Sven Reichard, Andrew Woldar, and Matan Ziv-Av. Examples of computer experimentation in algebraic combinatorics. *Ars Math. Contemp.*, 3(2):237–258, 2010.

[15] Mikhail H. Klin and Andrew J. Woldar. Dale Mesner, Higman & Sims, and the strongly regular graph with parameters $(100, 22, 0, 6)$. *Bull. Inst. Combin. Appl.*, 63:13–35, 2011.

[16] Martin Mačaj and Jozef Širáň. Search for properties of the missing Moore graph. *Linear Algebra Appl.*, 432(9):2381–2398, 2010.

[17] Spyros S. Magliveras. The subgroup structure of the Higman-Sims simple group. *Bull. Amer. Math. Soc.*, 77:535–539, 1971.

[18] B. D. McKay. nauty user's guide (version 1.5), 1990.

[19] Dale Marsh Mesner. *An investigation of certain combinatorial properties of partially balanced incomplete block experimental designs and association schemes, with a detailed study of designs of latin square and related types*. ProQuest LLC, Ann Arbor, MI, 1956. Thesis (Ph.D.)–Michigan State University.

[20] Dale Marsh Mesner. Negative latin square designs. Mimeo notes 410, Institute of Statistics, UNC, NC, 1964.

[21] George Neil Robertson. *Graphs minimal under girth, valency and connectivity constraints*. ProQuest LLC, Ann Arbor, MI, 1969. Thesis (Ph.D.)–University of Waterloo (Canada).

[22] Martin Schönert et al. *GAP – Groups, Algorithms, and Programming – version 3 release 4 patchlevel 4*. Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1997.

[23] Leonard H. Soicher. GRAPE: a system for computing with graphs and groups. In *Groups and computation (New Brunswick, NJ, 1991)*, volume 11 of *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.*, pages 287–291. Amer. Math. Soc., Providence, RI, 1993.

[24] Edwin R. van Dam. Three-class association schemes. *J. Algebraic Combin.*, 10(1):69–107, 1999.

[25] B. Weisfeiler. *On construction and identification of graphs*. Number 558 in Lecture Notes in Math. Springer, Berlin, 1976.

[26] Robert Wilson, Peter Walsh, Jonathan Tripp, Ibrahim Suleiman, Richard Parker, Simon Norton, Simon Nickerson, Steve Linton, John Bray, and Rachel Abbott. Atlas of finite group representations - version 3. `http://brauer.maths.qmul.ac.uk/Atlas/v3/`.

[27] M. Ziv-Av. Protocols with computer data related to the investigation of known primitive trangle-free strongly regular graphs. `http://www.math.bgu.ac.il/~zivav/math/`.

[28] M. Ziv-Av. Results of computer algebra calculations for triangle free strongly regular graphs. `http://www.math.bgu.ac.il/~zivav/math/eqpart.pdf`.

A Data about embeddings of tfSRGs inside tfSRGs

	Quadrangle	edge	2 edges	3	4	5	6
Pentagon	0	5	0	0	0	0	0
Petersen	0	15	15	5	0	0	0
Clebsch	40	40	60	40	10	0	0
HoSi	0	175	7875	128625	845250	2170350	1817550
Gewirtz	630	280	15120	245280	1370880	2603664	1643040
Mesner	6930	616	55440	1330560	10589040	28961856	24641232
$NL_2(10)$	28875	1100	154000	5544000	67452000	301593600	477338400

	7 edges	8	9	10	11
HoSi	40150	15750	3500	350	0
Gewirtz	104160	7560	1400	112	0
Mesner	3664320	166320	30800	2464	0
$NL_2(10)$	258192000	14322000	924000	154000	11200

Table 1: Number of imprimitive tfSRGs inside tfSRGs

	Quadrangle	edge	2 edges	3	4	5	6
Pentagon	0	1	0	0	0	0	0
Petersen	0	1	1	1	0	0	0
Clebsch	1	1	1	1	1	0	0
HoSi	0	1	1	4	10	21	15
Gewirtz	1	1	2	9	30	48	36
Mesner	1	1	1	7	26	56	50
$NL_2(10)$	1	1	1	2	7	14	17

	7 edges	8	9	10	11
HoSi	8	1	1	1	0
Gewirtz	5	2	2	1	0
Mesner	14	2	2	1	0
$NL_2(10)$	14	3	2	2	1

Table 2: Number of orbits of imprimitive tfSRGs inside tfSRGs

	Pentagon	Petersen	Clebsch	HoSi	Gewirtz	Mesner	$NL_2(10)$
Pentagon	1	12	192	1260	8064	88704	443520
Petersen		1	16	525	13440	1921920	35481600
Clebsch			1	0	0	0	924000
HoSi				1	0	0	704
Gewirtz					1	22	1030
Mesner						1	100
$NL_2(10)$							1

Table 3: Number of primitive tfSRGs inside tfSRGs

	Pentagon	Petersen	Clebsch	HoSi	Gewirtz	Mesner	$NL_2(10)$
Pentagon	1	1	1	1	1	1	1
Petersen		1	1	1	1	9	5
Clebsch			1	0	0	0	1
HoSi				1	0	0	1
Gewirtz					1	1	1
Mesner						1	1
$NL_2(10)$							1

Table 4: Number of orbits of primitive tfSRGs inside tfSRGs

B Data about equitable partitions of primitive tfSRGs

	Pentagon	Petersen	Clebsch	HoSi	Gewirtz	Mesner	$NL_2(10)$
EP	3	11	46	163			
Aut	3	11	38	89	154	236	607

Table 5: Number of orbits of equitable partitions and of automorphic equitable partitions for known tfSRGs.

Size	1	3	5	Total
EP	1	1	1	3
Aut	1	1	1	3

Size	1	2	3	4	5	6	7	10	Total
EP	1	2	2	2	1	1	1	1	11
Aut	1	2	2	2	1	1	1	1	11

Size	1	2	3	4	5	6	7	8	9	10	12	16	Total
EP	1	4	6	12	5	7	3	4	1	1	1	1	46
Aut	1	4	5	10	3	5	2	4	1	1	1	1	38

Size	1	2	3	4	5	6	7	8	9	10	11
EP	1	6	8	16	18	20	19	18	11	11	8
Aut	1	4	5	7	6	9	9	11	4	9	4
Size	12	13	14	15	16	18	20	28	30	50	Total
EP	7	7	2	2	1	2	3	1	1	1	163
Aut	4	4	2	1	1	2	3	1	1	1	89

Table 6: Numbers of EPs and automorphic EPs by size of partition for four small graphs.

Size	1	2	3	4	5	6	7	8	9	10	11	12	13
Aut	1	5	9	12	12	14	15	16	14	11	7	7	6
Size	14	16	17	18	19	20	23	28	31	32	35	56	Total
Aut	5	3	3	2	2	4	1	1	1	1	1	1	154

Size	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Aut	1	3	5	8	11	10	20	14	19	12	18	12	16	9	14
Size	16	17	18	19	20	21	23	25	29	33	41	45	49	77	Total
Aut	5	12	5	9	1	8	4	8	6	2	1	1	1	1	236

Size	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Aut	1	6	15	21	28	29	31	42	34	35	37	49	30	31	27
Size	16	17	18	19	20	21	22	23	24	25	26	27	28	29	
Aut	26	18	18	13	26	14	11	7	9	6	6	5	2	1	
Size	30	31	32	33	34	35	39	40	45	50	53	60	65	100	Total
Aut	7	1	3	2	2	3	1	4	1	1	1	1	1	1	607

Table 7: Numbers of automorphic EPs by size of partition for larger graphs.

Usage of Invariants for Symbolic Verification of Requirements

Short Paper

Alexander Letichevsky[1], Alexander Godlevsky[1], Anton Guba[1], Alexander Kolchin[1], Oleksandr Letychevskyi[1], Vladimir Peschanenko[2]

[1] V.M. Glushkov Institute of Cybernetics, Kiev, Ukraine
let@cyfra.net, godl@iss.org.ua, antonguba@ukr.net, kolchin_av@yahoo.com, lit@iss.org.ua
[2] Kherson State University, Kherson, Ukraine
vladimirius@gmail.com

The main goal of the paper is finding of pre- and post-invariants for transitions between symbolic states in the system that must be verified and use them for verification purposes. Systems are specified by basic protocols [1]. This specification defines a transition system with transitions $s \to s'$ where s and s' are symbolic states, b is a basic protocol. The main problem of verification is a reachability problem for given properties expressed in specification logic language.

We present double approximation method of verification based on computing invariants. The method is implemented as an iterative algorithm which solves in parallel reachability problem and computes lower and upper approximations of invariants for basic protocols.

Currently there are a lot of significant works devoted to computing invariants, which are considered for loops in programs. Loops are marked explicitly - syntactic constructions like while and others. When we deal with requirement specifications, we do not deal with explicit marked loops, moreover basic protocols provide non-deterministic order. Existing methods usually ignore the conditions of the loops. Invariants formulae, which is obtained as a result, include unreachable states and therefore could be used in verification only in the following way: if a property does not intersect with the invariant formula, then it is unreachable, if intersects, then conclusion cannot be done. Therefore, the problem of modification of existing methods, or to develop a new algorithm that could be applied in practice of requirements verification is actual.

Research of problems of automatic program invariants generation for a variety of data algebras was performed starting from 70-th years in Institute of Cybernetics of NAS of Ukraine. Their main results are presented in [2]. Double approximation method, is the dynamic iterative method of invariants generation and it is based on these results and adapted for requirements verification. The method also can be applied to program verification if a program is considered as a special case of basic protocol specification. The method has been implemented in VRS (Verification of Requirement Specifications) system [3] and IMS (Insertion Modeling System) system [4].

References

[1] A. Letichevsky, J. Kapitonova, V. Volkov, A. Letichevsky Jr., S. Baranov, V. Kotlyarov, T. Weigert. System Specification with Basic Protocols. Cybernetics and System Analyses, vol. 4, 2005, p. 3-21.

[2] Godlevsky A.B., Kapitonova Y.V., Krivoy S.L., Letichevsky A.A. Iterative methods of program analysis. Cybernetics, vol. 2, 1989, . 9-19.

[3] Verification for Requirement Specification (VRS). http://iss.org.uaISS/VRS/tool.htm, last viewed May 2013.

[4] APS and IMS Systems. http://apsystem.org.ua, last viewed May 2013.

Lebesgue Constants and Optimal Node Systems via Symbolic Computations

Short Paper

Robert Vajda

University of Szeged, Szeged, Hungary
vajdar@math.u-szeged.hu

Abstract

Polynomial interpolation is a classical method to approximate continuous functions by polynomials. To measure the correctness of the approximation, Lebesgue constants are introduced. For a given node system $X^{(n+1)} = \{x_1 < \ldots < x_{n+1}\}$ $(x_j \in [a, b])$, the Lebesgue function $\lambda_n(x)$ is the sum of the modulus of the Lagrange basis polynomials built on $X^{(n+1)}$. The Lebesgue constant Λ_n assigned to the function $\lambda_n(x)$ is its maximum over $[a, b]$. The Lebesgue constant bounds the interpolation error, i.e., the interpolation polynomial is at most $(1 + \Lambda_n)$ times worse then the best approximation. The minimum of the Λ_n's for fixed n and interval $[a, b]$ is called the optimal Lebesgue constant Λ_n^*. For specific interpolation node systems such as the equidistant system, numerical results for the Lebesgue constants Λ_n and their asymptotic behavior are known [3, 7]. However, to give explicit symbolic expression for the minimal Lebesgue constant Λ_n^* is computationally difficult. In this work, motivated by Rack [5, 6], we are interested for expressing the minimal Lebesgue constants symbolically on $[-1, 1]$ and we are also looking for the characterization of the those node systems which realize the minimal Lebesgue constants. We exploited the equioscillation property of the Lebesgue function [4] and used quantifier elimination and Groebner Basis as tools [1, 2]. Most of the computation is done in Mathematica [8].

Acknowledgement. The research of the author was partially supported by the HSRF (OTKA), grant number K83219.

References

[1] D. S. Arnon - G. E. Collins - S. McCallum, Cylindrical Algebraic Decomposition I: The Basic Algorithm. In: Caviness-Johnson (eds): Quantifier Elimination and Cylindrical Algebraic Decomposition, 136-151, Springer, 1998.

[2] B. Buchberger, Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal (An algorithm for finding the basis elements in the residue class ring modulo a zero dimensional polynomial ideal). PhD Thesis, Innsbruck, 1965.

[3] J. S. Hesthaven, From electrostatics to almost optimal nodal sets for polynomial interpolation in a simplex, SIAM J. Numer. Anal., Vol. 35, No. 2, 655-676, 1998.

[4] T. A. Kilgore, A characterization of the Lagrange interpolating projection with minimal Tchebycheff norm, J. Approx. Theory 24 (1978), no. 4, 273-288.

[5] H.-J. Rack, An example of optimal nodes for interpolation, International Journal of Mathematical Education in Science and Technology 15 (3): 355-357, 1984.

[6] H.-J. Rack, An example of optimal nodes for interpolation revisited, Advances in Applied Mathematics and Approximation Theory, Springer Proceedings in Mathematics & Statistics, Volume 41, 117-120, 2013.

[7] S. J. Simon, Lebesgue constants in polynomial interpolation, Annales Mathematicae et Informaticae 33: 109-123, 2006.

[8] Wolfram Research Inc., Mathematica, Version 9.0, Champaign, Illinois, 2012.

Author Index

Keyword Index