Verification Techniques for Extensions of Equality Logic

Bahareh Badban

© Bahareh Badban, Amsterdam 2006
Printed by Ponsen & Looijen
ISBN 90-6196-535-7
IPA dissertation series 2006-13

All rights reserved, including the right of reproduction in whole or in part in any form.

The research in this thesis has been carried out at the Centre for Mathematics and Computer Science (CWI), under the auspices of the research school IPA (Institute for Programming research and Algorithmics). The research was carried out in a project granted by the Dutch organization for Scientific Research (NWO), on "Integrating Techniques for the Verification of Distributed Systems".

VRIJE UNIVERSITEIT

Verification Techniques for Extensions of Equality Logic

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op donderdag 7 september 2006 om 13.45 uur
in het aula van de universiteit,
De Boelelaan 1105

door

Bahareh Badban

geboren te Tehran, Iran

promotor: prof.dr. W.J. Fokkink
copromotor: dr. J.C. van de Pol

Acknowledgments

I studied four years at CWI. These years included breakthroughs and breakdowns in the research I was doing. I present in this book a collection of most of the breakthroughs and a few breakdowns as well.

Many people have been involved in my success, some in the academic environment and some in my personal life in Amsterdam. I am grateful to all those who were part of my life. Meeting them helped me to make a life that I liked.

Here I would like to thank my academic partners: my co-promotor Jaco van de Pol has been the closest working partner to me through all these experiences. Whenever we faced a failure he would smile and keep looking toward success. I thank him for his lots of care, and his intellectual ideas.

I thank my promoter Wan Fokkink for his warm and productive presence at CWI. I deeply appreciate his great support in accomplishing the second part of my thesis. He guided me through the time I was preparing the manuscript.

Hans Zantema, Olga Tveretina, Bas Luttik from the Technical University of Eindhoven, and Jun Pang from CWI, have been the coauthors for parts of the work. I enjoyed our contribution and our fruitful discussions.

I acknowledge the reading committee of my thesis, alphabetically: Jan Bergstra, Jan Willem Klop, Bas Luttik and Hans Zantema, for their constructive comments and questions.

Anton Wijs took it on to translate the summary to Dutch. I thank him for doing so. He was a very friendly, caring officemate. I had a great time with my colleagues at CWI, those who left and those who are still at CWI. I thank Stefan Blom, Yaroslav Usenko, Natalia Ioustinova, Jun Pang, Simona Orzan and Miguel Valero whose experiences I benefited from in the process of writing my thesis manuscript.

I am thankful to my parents and my family from the bottom of my heart for their lots of love and support. Their care and encouragements have been always with me.

Bahareh

Contents

1 Introduction **1**

I Decision Procedures for Extensions of Equality Logic **13**

2 A Complete Method for BDDs with $(0, S, =)$ **15**
- 2.1 Introduction 15
- 2.2 Binary Decision Diagrams 16
- 2.3 BDDs with Equality, Zero and Successor 17
- 2.4 Elimination-Ordered $(0, S, =)$-BDDs 18
- 2.5 Conclusion 39

3 Representant-Ordered $(0, S, =)$-BDDs **43**
- 3.1 Introduction 43
- 3.2 Representant-Ordered $(0, S, =)$-BDDs 44
- 3.3 An Algorithm 55
- 3.4 Failed Attempts 62
- 3.5 Conclusion 65

4 Ground Term Algebra **67**
- 4.1 Introduction 67
- 4.2 Preliminaries 68
- 4.3 Ground Term Algebra 71
- 4.4 Substitutions and Most General Unifiers 72
- 4.5 GDPLL 74
- 4.6 Termination 78
- 4.7 Correctness Properties 81
- 4.8 Correctness of GDPLL 84
- 4.9 The Witness 85
- 4.10 Comparison, OBDD and GDPLL methods 90
- 4.11 Conclusion 90

5 Ground Term Algebra with Recognizers 93
5.1 Introduction . . . 93
5.2 Recognizers . . . 94
5.3 Transformation . . . 96
5.4 Decision Procedure . . . 103
5.5 Lists and List Operations . . . 104
5.6 Conclusion . . . 105

II Verification of Protocols 107

6 Mechanical Verification of a Two-Way Sliding Window Protocol 109
6.1 Introduction . . . 109
6.2 μCRL . . . 110
6.3 Data Types . . . 115
6.4 A Two-Way Sliding Window Protocol with Piggybacking . . . 120
6.5 Transformations of the Specification . . . 124
6.6 Properties of Data Types . . . 129
6.7 Correctness of $\mathbf{N}_{mod}$ . . . 133

7 Formalization and Verification in PVS 147
7.1 Data Specifications in PVS . . . 148
7.2 Representing LPEs . . . 151
7.3 Representing Invariants . . . 154
7.4 Equality of $\mathbf{N}_{mod}$ and $\mathbf{N}_{nonmod}$ . . . 154
7.5 Correctness of $\mathbf{N}_{mod}$ . . . 154
7.6 Remarks on the Verification in PVS . . . 155

A Proofs on Properties of Data 157

Summary 197

Samenvatting 199

1

Introduction

One of the most important subfields of automated reasoning is automated theorem proving. Automated theorem proving means proving mathematical theorems by a computer program. Depending on the underlying logic, the problem of deciding the validity of a theorem varies from trivial to impossible. For propositional logic, the problem is decidable. For first-order logic with equality, it is semi-decidable (recursively enumerable). This means that given unbounded domains, any valid theorem can eventually be proven but invalid theorems cannot always be recognized. In these cases a first-order theorem prover might run forever. Although, the decision problem for plain equational theories in general is unsolvable, yet first-order theorem proving is a mature subfield of automated theorem proving. The logic is quite expressive to solve many hard problems in a natural way.

The existing automated theorem proving techniques can be roughly classified into two main categories *encoding* and *incremental* methods. The encoding method is also divided to *eager* or *lazy* encoding. In the eager approach the input formula is translated into an equivalent *Conjunctive Normal Form* (CNF) propositional formula. Then the result is checked with an existing SAT solver (i.e. a propositional theorem prover). UCLID is an example of this method. In the lazy approach each atom of the input formula is replaced by a distinct propositional variable. Then a SAT solver is used to find an assignment for the propositional formula which does not satisfy it. Such an assignment will also dissatisfy the original formula. Hence this assignment is later on discarded by adding a proper statement to the original formula. This process is repeated until a satisfying assignment is obtained or all the assignments have been dismissed. Some recently developed theorem provers like SVC [BDL96], CVC [SBD02] and ICS [SR02, FORS01] are of this kind. These theorem provers were built recently in Stanford. CVC has lately been replaced with a stronger theorem prover called CVC lite [BB04]. Though CVC lite verifies a very rich sub-theory of

first-order logic, yet it does not include algebraic data types (i.e. those types which include one or more constructors).

The second category includes the methods in which an existing theorem prover has been extended to a new theorem prover over a larger logic. One of the advantages of these methods is that they can be directly applied on the input formulas without any need for transformation to a propositional statement. Examples of this are the BDD method extension from propositional logic to equality logic by Groote and van de Pol in [GvdP00], and also the DPLL(T) approach which is introduced by Tinelli in [Tin02].

During verification of distributed systems many proof obligations on data are generated. Proving such formulas automatically, or dually verifying unsatisfiability of their negation, is essential for large distributed systems. For many theories, decision procedures for deciding satisfiability of conjunctions of (negated) equations exist. Examples include linear and integer programming for arithmetic over integers or reals, congruence closure algorithms to deal with uninterpreted functions (i.e. second-order variables) -this research was initiated by Shostak [Sho78] and Nelson and Oppen [NO80]- and the Fourier-Motzkin transformation [BW94] for dealing with linear inequalities. Having this need, the first part of the thesis is dedicated to finding an answer to the question to what extent we are able to automate the verification of first-order theories. Our approach stands in the second category. In the first part of the thesis we extend two of the most popular existing theorem provers, Binary Decision Diagrams and DPLL techniques, to larger theories.

Another group of theorem provers called interactive theorem provers, require a human user to give hints to the system. Depending on the degree of automation, the prover can essentially be limited to a proof checker, with the user providing the proof in a formal way, or significant proof tasks can be performed automatically. PVS is an example of such theorem provers. It is an environment for constructing clear and precise specifications and for developing readable proofs that have been mechanically verified [ORR+96]. We use this theorem prover to establish the reliability of a protocol which is part of most people's daily lives these days.

To date, *Transmission Control Protocol* (TCP) is used in roughly all Internet applications. Some well-known ones that use TCP are HTTP/HTTPS for World Wide Web, SMTP/IMAP/POP3 for email and FTP for file transfer. TCP implementations have been optimized for use in wired and wireless networks [Tan81]. Because of the widespread use of TCPs many areas of research including enhancing TCP to reliably handle loss of data, minimize errors, manage congestion, go fast in very high-speed environments and have more efficient use of available bandwidth networks are going on. This protocol is also supposed to guarantee reliable and in-order data delivery of sender to receiver and vice versa. The sliding window protocol is the protocol that provides handling timeouts and also transmissions/retransmissions of data in TCP. Since TCP manages retransmission of data if it is not received within a reasonable

round-trip time, hence a reliable and efficient sliding window protocol can produce a TCP with a better use of the network bandwidth. In the second part of the thesis we first present a specification of a sliding window protocol with *piggybacking*. Then we use the capability of a theorem prover in order to obtain a reliable protocol. In the last chapter we verify the provided specification of Chapter 6, in PVS theorem prover. This provides a totally reliable protocol.

Overview of the thesis

Part I. Decision Procedures for Extensions of Equality Logic

Chapters 2, 3, 4 and 5 focus on interpreted extensions of equality logic. Two different approaches are introduced in these chapters. In Chapters 2 and 3 we extend the well-known binary decision diagrams (BDDs) to the theory of first-order logic with zero, successor and equality. We use two approaches based on two different orders on equations. Then we present an algorithm for the verification of our formulas. In Chapters 4 and 5 we take another approach based on the Davis-Putnam-Logemann-Loveland (DPLL) method. This procedure has recently become very popular because of its simplicity and effectiveness in propositional logic [MMZ$^+$01]. With our generalized DPLL procedure we are able to decide the theory of Equality Logic with Constructors and Recognizers. A visible difference between these two techniques is that the BDD method is applicable on arbitrary formulas of its underlying logic, while the DPLL method only works with CNF formulas.

Groote and Zantema in [GZ03] provide examples in propositional logic which indicate that the BDD and resolution based techniques (e.g. DPLL) are different in essence. Theses two techniques perform quite dissimilar on benchmarks. They show how either of these two techniques can at a time outperform the other. As an example they prove that the class of propositional pigeon hole formulas (i.e. $\phi_n = \bigwedge_{1\leq i<j\leq n} \neg(p_i \leftrightarrow p_j) \wedge \bigwedge_{1\leq j\leq n}(\bigvee_{1\leq i\leq n,\ i\neq j} p_i \leftrightarrow q))$ are easy to be verified by resolution based methods like DPLL but exponentially hard for Ordered BDDs. On the other hand biconditional propositional formulas (i.e. the formula only include $\leftrightarrow$ and $\neg$ as binary symbols) are easy for Ordered BDDs but exponentially hard for DPLLs.

Extension of Ordered BDDs

In this section I explain the methods in the two first chapters. The methods are based on extensions of Ordered BDDs.

In Chapters 2 and 3 we investigate the satisfiability and tautology problem for boolean combinations over the equational theory (i.e. quantifier-free first order logic with equality as the only predicate symbol) of zero and successor in the natural numbers. Chapter 2 is based on [BvdP05] and Chapter 3 on [BvdP04]. Chapter 3 is an extension of the approach in [GvdP00] to a larger logic. Groote and van de Pol in [GvdP00] introduced an ordering technique for equational first-order logic, without function symbols. The extension presented in Chapter 3 might seem trivial, but in fact this is not the case. As an evidence for this claim, I explain two extensions in Section 3.4 which might look quite decent but they fail in reaching an Ordered BDD form for certain formulas.

Chapter 2 introduces a proper ordering which results in having Ordered BDDs for any formula of its underlying logic. Chapter 3 presents a totally new ordering. As a result different sets of Ordered BDDs will be obtained. Here we also make an algorithm based on the defined term rewrite system.

These two methods share a common purpose. They both aim at presenting a structure to transform any statement of their (join) logic, to an equivalent Ordered BDD in a way that tautology and satisfiability can be checked in constant time.

On the other hand, these two techniques are not symmetric and the completeness of one does not result in the completeness of the other. Moreover they are different in being capable of upgrading to larger theories. The method in Chapter 2 is later on extended to a decision procedure for Equality Logic with Uninterpreted Functions by van de Pol and Tveretina [vdPT05] while Chapter 3 fails in that. There are far more theories, containing our equational theory, which are the basis of many existing theorem provers. Whether or not our approach of Chapter 3 can be extended further to larger theories, cannot be foretold at this stage. However the method by itself is a new theorem prover in a quite expressive logic. Moreover it does provide a second opportunity in the research area of developing the existing procedures (as opposed to encoding methods) to larger theories.

In these two chapters atoms are equations between terms built from variables, zero (0) and successor (S). Formulas are built from atoms by means of negation ($\neg$) and conjunction ($\wedge$). The formulas are quantifier-free, except for the implicit outermost quantifier ($\forall$ when considering tautology checking, and $\exists$ when considering satisfiability).

We shortly review what we will call the DNF method, the plain BDD method, the Encoding method and the EQ-BDD method.

In the DNF method, the first-order formula is transformed to a propositionally equivalent *Disjunctive Normal Form* (DNF). A formula in DNF is satisfiable if and only if at least one of its disjuncts is satisfiable. Such a disjunct is a conjunction of literals (equations and negated equations).

The DNF method has a clear bottleneck, because the transformation to DNF may result in a formula that is exponentially larger than the original. This is improved by

the plain BDD method. In that method, a formula is transformed to a propositionally equivalent *Ordered Binary Decision Diagram* (OBDD [Bry92]), which is a binary directed acyclic graph. Each node is labeled with an atomic proposition, and has a left and a right descendant. The leaves can be either $\top$ (true) or $\bot$ (false). A BDD can be viewed as an if-then-else (*ITE*) tree with shared sub-terms, where the tests are atomic propositions. In an OBDD, the order of the tests in each path of this tree is fixed by a total order on atoms. Although OBDD representations can also be exponentially large, it appears that in practice many formulas have a succinct OBDD-representation.

Two propositionally equivalent OBDDs are identical. This means that if all atoms are propositional symbols, then OBDDs are unique representations of boolean functions. However, in our case the atoms are equations, and uniqueness is lost. For instance, the OBDDs $ITE(x = y, \top, \bot)$ and $ITE(y = x, \top, \bot)$ are equivalent, although not syntactically equivalent. Similarly, in the propositional case all paths in an OBDD represent a satisfiable conjunction, but in the equational case this property is lost. For instance, the path to $\bot$ in $ITE(x = y, ITE(y = x, \top, \bot), \top)$ represents the inconsistent conjunction $x = y \wedge y \neq x$. As a result, OBDDs with $\bot$-leaves can still be a tautology.

In order to solve the satisfiability or tautology problem using OBDDs, it must be checked for each path in the OBDD whether it represents a consistent conjunction with respect to the underlying equational theory. This is done by applying the aforementioned decision procedures. If all "consistent" paths lead to $\top$-leaves, then the OBDD is a tautology. If all consistent paths lead to $\bot$-leaves, then the OBDD is a contradiction. Otherwise, it is just satisfiable. This procedure is both sound and complete, but due to sharing of sub-terms, an OBDD can have exponentially many paths, so still there is a computational bottleneck. A typical example of this approach are the DDDs (difference decision diagrams) of [MLAH99], where atoms can be of the form $x < y + c$, for variables x and y and a constant c (known as separation predicates [Pra70], or difference logic). So here having a technique which can remove the unsatisfiable paths emerges. In our approach all paths in an OBDD ($(0, S, =)$-OBDD) are satisfiable (i.e. consistent).

In both the DNF and the plain BDD method, the boolean structure is flattened out immediately, and the arithmetic part is dealt with in a second step. In the *Encoding method* these steps are reversed. First the formula is transformed to a purely propositional formula, which is satisfiable if and only if the original formula is satisfiable in the equational theory. In this translation, facts from the equational theory (e.g. congruence of functions, transitivity of equality and orderings) are encoded into the formula. Then a *finite model property* is used to obtain a finite upper bound on the cardinality of the model. Finally, variables that range over a set of size n are encoded by $log(n)$ propositional variables. The resulting formula can be checked for satisfiability with any existing procedure for deciding satisfiability of propositional formulas,

for instance based on resolution or on propositional BDDs. An early example is Ackermann's reduction [Ack54], by which second-order variables can be eliminated. More optimal versions can be found in [GSZA98, PRSS99, BGV01]. Recently, this method is applied in [SLB02] to boolean combinations over successor, predecessor, equality and inequality over the integers; in [SSB02] it has been applied to separation predicates $x < y + c$; and in [Str02], Pressburger arithmetic for integers and linear arithmetic for reals are translated into propositional logic.

In the last approach that we mention, called the EQ-BDD method (*Binary Decision Diagrams extended with Equality* [GvdP00]), boolean and arithmetic reasoning are not separated, but intertwined. Similar to the plain BDD approach, an ordered EQ-BDD is constructed, but during this construction, facts from the equational theory are used to prune inconsistent paths at an earlier stage. The main technique is a substitution rule, which allows to replace $ITE(s = t, \varphi(s), \psi)$ by $ITE(s = t, \varphi(t), \psi)$. It was shown that an equivalent normal form can always be found, and they have the desirable property that all paths in it represent consistent conjunctions. As a consequence, $\top$ and $\bot$ have a unique EQ-OBDD representation, so tautology, contradiction and satisfiability checking on EQ-OBDDs can be done in constant time. The resulting EQ-OBDDs are logically equivalent to the original formula (not just equi-satisfiable, as in the translations to propositional logic), so this technique can also be used to simplify a given formula. It is this last approach that we have extended with zero and successor.

Generalization of DPLL

Below we introduce the method we present in Chapters 4 and 5, which is based on a generalization of the DPLL procedure. Chapter 4 is based on [BvdPTZ04a]. Chapter 5 presents new results.

In Chapters 4 and 5 we provide a generalization of the well-known DPLL procedure, named after Davis-Putnam-Logemann-Loveland. DPLL [DP60, DLL62] has been mainly used to decide satisfiability of propositional formulas, represented in CNF. The main idea of this recursive procedure is to choose an atom from the formula and proceed with two recursive calls: one for the formula obtained by adding this atom as a fact and one for the formula obtained by adding the negation of this atom as a fact. Intermediate formulas may be further reduced. The search terminates as soon as a satisfying assignment is found, or alternatively, a satisfiability criterion may be used to terminate the search. This idea may be applied to other kinds of logics too. We will focus on certain quantifier-free fragments of first-order logic for which this yields a (terminating) sound and complete decision procedure for satisfiability.

We provide a concrete algorithm for the quantifier-free equational logic over the ground term algebra and over ground term algebra with recognizers (a certain kind of predicates). In Chapter 4 we give an algorithm (called GDPLL) for ground term

algebra out of which a witness for any satisfiable formula can be obtained. In Chapter 5 we give a further generalization of the DPLL algorithm (called RGDPLL) to ground term algebra with recognizers. The idea is to transform every formula involving recognizers to a formula without recognizers (i.e. in ground term algebra) and then check its satisfiability using the GDPLL algorithm.

Our main motivation has been to decide boolean combinations over algebraic data types. In many algebraic systems, function symbols are divided in constructors and defined operations. The values of the intended domains coincide with the ground terms built from constructor symbols only. This is for instance the case with the data specifications in μCRL [GR01, BFG$^+$01], a language based on abstract data types and process algebra.

Our algorithm works for constructor symbols (examples: zero, succ, cons and nil) and recognizer predicates (standard ones such as nil?, succ?, cons?, zero?).

Our tool is comparable to ICS [SR02, FORS01] (which is used in PVS), CVC [BDS00, SBD02] and CVC-lite [BB04], but as opposed to these tools, our algorithm is sound and complete for the ground term algebra. The ICS and CVC tools combine several decision procedures based on an algorithm devised by Shostak. Among these are a congruence closure algorithm for uninterpreted functions, and a decision procedure for arithmetic, including + and >. They also support abstract data types. In ICS abstract data types are specified as a combination of products and co-products; in CVC abstract data types can be defined inductively. ICS and CVC tools both are incomplete for quantifier-free logic over abstract data types. For instance, experiments show that CVC does not prove validity of the query $x \neq succ(succ(x))$. Algebraic data types are not yet incorporated in CVC lite.

FDPLL [Bau00] is a generalization of DPLL to first-order logic. Note that FDPLL solves a different problem. First, it deals with quantifiers. Second, it does not take into account equality, or fixed theories, such as ground term algebras. The algorithm is called sound and complete, but it is not terminating, because satisfiability for first-order logic is undecidable. Our GDPLL is meant for decidable fragments, so we only deal with quantifier-free logics.

Recall that other approaches encode the satisfiability question for a particular theory to plain propositional logic. For the logic of equality and uninterpreted function symbols, one can use Ackermann's reduction [Ack54, BGV99] to transform this logic to propositional logic, so any propositional logic satisfiability checker can be used.

Our particular solution for ground term algebras depends on a well-known unification theory. Ground breaking work in this area was done by [Rob65]. Unification solves conjunctions of equations in the ground term algebra. Colmerauer [Col84] studied a setting with conjunctions of both equations and inequations. Using a DNF transformation, this is sufficient to solve any boolean combination. However, the DNF transformation itself may cause an exponential blow-up. For this reason we base our algorithm on DPLL, where after each case split the resulting CNFs can be reduced

(also known as constraint propagation). In particular, our reduction is based on a combination of unification and unit resolution which states that a literal appearing in a "unit clause" must be assigned true.

For an extensive treatment of unification see [LMM87], and for a textbook on unification (theory and algorithms) we recommend [BN98]. The full first-order theory of equality in ground term algebras is studied in [Mah88, CL89] (both focus on a complete set of rewrite rules) and more recently by [Pic03] (who focuses on complexity results for DNFs and CNFs in case of bounded and unbounded domains). Our algorithm is consistent with Pichler's conclusion that for unbounded domains the transformation to CNF makes sense. Very recently the DPLL(T) approach was introduced for satisfiability of modulo theories T [Tin02].

Recently Ganzinger et al. used the DPLL(T) calculus ([GHN+04]) to provide a new approach for satisfiability in the EUF logic and also EUF logic with successor and predecessor. None of these papers give concrete algorithms for use in verification, and the idea to combine unification and DPLL seems to be new [BvdPTZ04a].

Part II. Verification of Protocols

In Chapters 6 and 7 we focus more on application, by verifying a two-way sliding window protocol. We use μCRL as the specification language and then we verify all the theorems and propositions in PVS. Chapter 6 presents the details of verification in μCRL. In Chapter 7 we bring some summary on how the verification of the whole theory is done in PVS and as a witness we show some PVS code samples.

Mechanical Verification of a Two-Way Sliding Window Protocol

In Chapter 6 we prove the correctness of a two-way sliding window protocol (SWP) with piggybacking, for an arbitrary finite window size n and sequence numbers modulo $2n$. In a *two-way* SWP, both parties can both send and receive data elements to and from each other. The correctness consists of showing that this SWP is branching bisimilar [vGW96] to a pair of FIFO queues of capacity $2n$. We model the protocol and prove its correctness in the process algebraic language μCRL. The correctness proof has been formalized using the theorem prover PVS [ORR+96], described in Chapter 7.

This chapter builds on a verification of a one-way version of the SWP in [FGP+04, BFG+05], where the protocol was specified in μCRL [GP95], which is a language based on process algebra and abstract data types. That verification was also formalized in PVS. The correctness proof in [FGP+04, BFG+05], and also in the current chapter, is based on the so-called *cones and foci* method [GS01, FP03, FPvdP05], which rephrases

the question whether a protocol specification exhibits the desired external behaviour in terms of data equalities, called *matching criteria*.

We present a specification in μCRL of a two-way SWP with piggybacking, with buffer size $2n$ and window size n, for arbitrary n. The medium between the sender and the receiver is modeled as a lossy queue of capacity one. We manually prove that the external behaviour of this protocol is branching bisimilar to a pair of FIFO queues of capacity $2n$. This implies both safety and liveness of the protocol (the latter under the assumption of fairness).

SWPs have attracted considerable interest from the formal verification community. In this section we present an overview. Many of these verifications deal with unbounded sequence numbers, in which case modulo arithmetic is avoided, or with a fixed finite buffer and window size at the sender and the receiver.

Unbounded sequence numbers Stenning [Ste76] studied a SWP with unbounded sequence numbers and an infinite window size, in which messages can be lost, duplicated or reordered. A timeout mechanism is used to trigger retransmission. Stenning gave informal manual proofs of some safety properties. Knuth [Knu81] examined more general principles behind Stenning's protocol, and manually verified some safety properties. Hailpern [Hai82] used temporal logic to formulate safety and liveness properties for Stenning's protocol, and established their validity by informal reasoning. Jonsson [Jon87] also verified safety and liveness properties of the protocol, using temporal logic and a manual compositional verification technique. Rusu [Rus01] used the theorem prover PVS to verify safety and liveness properties for a SWP with unbounded sequence numbers.

Fixed finite window size Vaandrager [Vaa86], Groenveld [Gro87], van Wamel [vW92] and Bezem and Groote [BG94a] manually verified in process algebra a SWP with window size one. Richier *et al.* [RRSV87] specified a SWP in a process algebra based language Estelle/R, and verified safety properties for window size up to eight using the model checker Xesar.

Madelaine and Vergamini [MV91] specified a SWP in Lotos, with the help of the simulation environment Lite, and proved some safety properties for window size six.

Holzmann [Hol91, Hol97] used the Spin model checker to verify safety and liveness properties of a SWP with sequence numbers up to five.

Kaivola [Kai97] verified safety and liveness properties using model checking for a SWP with window size up to seven.

Godefroid and Long [GL99] specified a full duplex SWP in a guarded command language, and verified the protocol for window size two using a model checker based on Queue BDDs. Stahl *et al.* [SBLS99] used a combination of abstraction, data independence, compositional reasoning and model checking to verify safety and liveness properties for a SWP with window size up to sixteen. The protocol was specified in

Promela, the input language for the Spin model checker. Smith and Klarlund [SK00] specified a SWP in the high-level language IOA, and used the theorem prover MONA to verify a safety property for unbounded sequence numbers with window size up to 256.

Jonsson and Nilsson [JN00] used an automated reachability analysis to verify safety properties for a SWP with a receiving window of size one.

Latvala [Lat01] modeled a SWP using Coloured Petri nets. A liveness property was model checked with fairness constraints for window size up to eleven.

Arbitrary finite window size Cardell-Oliver [CO91] specified a SWP using higher order logic, and manually proved and mechanically checked safety properties using HOL. (Van de Snepscheut [vdS95] noted that what Cardell-Oliver claims to be a liveness property is in fact a safety property.) Schoone [Sch91] manually proved safety properties for several SWPs using assertional verification.

Van de Snepscheut [vdS95] gave a correctness proof of a SWP as a sequence of correctness preserving transformations of a sequential program. Paliwoda and Sanders [PS91] specified a reduced version of what they call a SWP (but which is in fact very similar to the bakery protocol from [GK95]) in the process algebra CSP, and verified a safety property modulo trace semantics.

Röckl and Esparza [RE99] verified the correctness of this bakery protocol modulo weak bisimilarity using Isabelle/HOL, by explicitly checking a bisimulation relation. Chkliaev *et al.* [CHdV03] used a timed state machine in PVS to specify a SWP with a timeout mechanism and proved some safety properties with the mechanical support of PVS; correctness is based on the timeout mechanism, which allows messages in the mediums to be reordered.

PVS

Chapter 7 shows the formalization and verification of the theory of Chapter 6 in PVS. The specification language of PVS is based on simply typed higher-order logics. PVS provides a rich set of types and the ability to define subtypes and dependent types. Though type checking is undecidable for the PVS type system, its type checker automatically checks for simple type correctness and generates proof obligations [ORR$^+$96]. It has a tool set consisting of a type checker, an interactive theorem prover, and a model checker. PVS has high-level proof strategies and decision procedures that take care of many low-level details associated with computer-aided theorem proving. In addition, PVS has useful proof management facilities, such as a graphical display of the proof tree, and proof stepping and editing.

PVS enabled us to find non-terminating definitions in the original data specification of Chapter 6, which were not detected within the framework of μCRL. In Section 7.1 we show some of the most interesting examples of this kind.

Conclusion

First-order logic is expressive enough to allow the specification of a wide range of problems often in a reasonably natural and intuitive way. First-order theorem proving is one of the most mature subjects of automated theorem proving. Although it is semi-decidable, a number of sound and complete calculi have been developed, which enable fully automated systems. Inside of this logic, proving the satisfiability of ground formulas (with no variables) is an important research problem with applications in many areas of computer science and artificial intelligence, such as software and hardware verification [Tin02].

In this thesis we extend two different decision procedure for subclasses of first-order logic, consisting of ground formulas. The extensive use of OBDDs in many areas such as digital-system design, verification, testing, concurrent-system design and artificial intelligence [Bry92], motivated us to extend the underlying logic of this decision procedure to the theory of equality with zero and successor. On the other hand most state-of-the-art propositional logic satisfiability checkers today are based on different variations of the DPPL procedure [NO05]. We also extend this decision procedure to the theory of ground term algebras with recognizers. We use our generalized DPLL procedure on LISP, as an instance of recursive data types. LISP is used extensively in programming languages.

In the end we verify a protocol. Sliding Window Protocols provide TCPs the ability of retransmission of data. They also take care of the timeouts for doing the retransmissions when data is lost. A more powerful and reliable SWP will certainly cause a stronger TCP, with fewer crashes. We apply the PVS theorem prover to verify that the μCRL specification of a two-way sliding window protocol with piggybacking is correct.

Part I

Decision Procedures for Extensions of Equality Logic

2

A Complete Method for BDDs with $(0, S, =)$

We extend BDDs (binary decision diagrams) for plain propositional logic to the fragment of first-order logic, consisting of quantifier-free logic with zero, successor and equality. We allow equations with zero and successor in the nodes of a BDD, and call such objects $(0, S, =)$-BDDs. We also extend the notion of *Ordered* BDDs in the presence of zero, successor and equality. $(0, S, =)$-BDDs can be transformed to equivalent *Elimination-Ordered* $(0, S, =)$-BDDs by applying a number of rewrite rules until a normal form is reached. The word "Elimination-Ordered"- as will be introduced in details later in this chapter- refers to the fact that we order the terms occurring on a BDD with respect to the variables which will are eliminated from the terms coming after them. All paths in these Elimination-Ordered $(0, S, =)$-BDDs represent satisfiable conjunctions. The major advantage of transforming a formula to an equivalent Elimination-Ordered $(0, S, =)$-BDD is that on the latter it can be observed in constant time whether the formula is a tautology, a contradiction, or just satisfiable, i.e. the time needed for its computation does not depend on the size of the given Elimination-Ordered $(0, S, =)$-BDD.

2.1 Introduction

BDDs (binary decision diagrams) represent boolean functions as directed acyclic graphs. BDDs are of value for validating formulas in propositional logic. Although BDDs are a great tool to verify formulas, a representative BDD of a formula can be exponentially large. In [Bry92] OBDDs (*Ordered* BDDs) are reduced BDDs which obey some ordering on boolean variables. A boolean function is satisfiable if and only if its unique OBDD representation does not correspond to $\perp$.

To date many attempts have been made to verify the satisfiability of different logics

using the BDD approach. With this intention, several methods have been proposed to reduce these logics into propositional logic, which captures boolean functions. Goel et al. [GSZA98] and Bryant et al. [BGV01] present methods to transform the logic of Equality with Uninterpreted Functions (EUF) into propositional logic.

What we do in Chapters 2 and 3 is in the opposite direction, which is extending the theory of BDDs, with the intention of enriching it to be able to verify more expressive logics directly, with no need for encoding techniques. The idea of extending the theory of BDDs was recognized earlier by Groote and van de Pol [GvdP00], who presented an algorithm to transform EQ-BDDs to EQ-OBDDs, where EQ-BDDs represent the extension of BDDs with equalities (without any function symbols). We make a terminating set of rewrite rules on $(0, S, =)$-BDDs, resulting in a $(0, S, =)$-E-OBDD (Elimination-Ordered $(0, S, =)$-BDD), such that all paths in the $(0, S, =)$-OBDD are satisfiable. This property enables us to check tautology, contradiction and satisfiability on $(0, S, =)$-E-OBDDs in constant time.

This chapter is based on [BvdP05] and is structured as follows. In Section 2.2 we first shortly introduce BDDs, then in Section 2.3 we present some preliminary notations and definitions with respect to the syntax and semantics of the formulas that we will deal with in Chapters 2 and 3. We also give a formal syntax and semantics of $(0, S, =)$-BDDs. In Section 2.4 $(0, S, =)$-E-OBDDs are defined. First any total and well-founded order on variables is extended to a total and well-founded order on atomic guards. Then the rewrite system is presented. In the end we prove termination and satisfiability over all paths. Finally, Section 2.5 concludes with some remarks on implementation and possible applications.

2.2 Binary Decision Diagrams

A *binary decision diagram* [Bry92] (BDD) represents a boolean function as a finite, rooted, binary, ordered, directed acyclic graph. The leaves of this graph are labeled $\bot$ and $\top$, and all internal nodes are labeled with boolean variables. A node with label p, left child L and right child R, written $ITE(p, L, R)$, represents the formula *if* p *then* L *else* R.

Given a fixed total order on the propositional variables, a BDD can be transformed to an *Ordered* binary decision diagram (OBDD), in which the propositions along all paths occur in increasing order, redundant tests ($ITE(p, x, x)$) don't occur, and the graph is maximally shared. For a fixed order, each boolean function is represented by a unique reduced OBDD (in the sequel we simply use OBDD to denote a reduced OBDD). Furthermore, boolean operations, such as negation and conjunction, can be computed on OBDDs very cheaply. Together with the fact that (due to sharing) many practical boolean functions have a small OBDD representation, OBDDs are very popular in verification of hardware design, and play a major role in symbolic model checking.

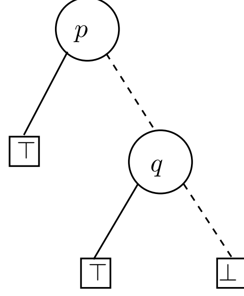

Figure 2.1: $ITE(p, \top, ITE(q, \top, \bot))$

Example 2.1 *Figure 2.1 illustrates the BDD representation of the formula* $ITE(p, \top, ITE(q, \top, \bot))$*, where* p *and* q *are propositional variables.*

2.3 BDDs with Equality, Zero and Successor

In this section we will introduce some basic notations and definitions which are essential in Chapter 2 and Chapter 3. We also provide the syntax and semantics of BDDs extended with zero, successor and equality. For our purpose, the sharing information present in the graph is immaterial, so we formalize BDDs by terms (i.e. trees). We view BDDs as a restricted subset of formulas, and show that every formula is representable as BDD.

Assume V is a set of variables, and define $\bar{V} = V \cup \{0\}$. We define sets of terms, formulas, guards and BDDs as follows.

Definition 2.2 *Terms* $t \in W$*, formulas* $\varphi \in \Phi$*, guards* $g \in G$ *and* $(0, S, =)$*-BDDs* $T \in B$ *are defined by the following grammar (with* $x \in V$*):*

$$
\begin{array}{rcl}
t & ::= & 0 \mid x \mid S(t) \\
\varphi & ::= & \bot \mid \top \mid t = t \mid \neg\varphi \mid \varphi \wedge \varphi \mid ITE(\varphi, \varphi, \varphi) \\
g & ::= & \bot \mid \top \mid t = t \\
T & ::= & \bot \mid \top \mid ITE(g, T, T)
\end{array}
$$

For technical reasons (see Definition 2.6) $\top$ and $\bot$ are allowed as guards.

We now introduce some notational conventions. Throughout this chapter $\equiv$ is used to denote syntactic equality between terms or formulas, in order to avoid confusion with the $=$-symbol in guards. Symbols $x, y, z, u, \ldots$ denote variables; $r, s, t, \ldots$ range over W; $\varphi, \psi, \ldots$ range over Φ; $f, g, \ldots$ range over guards. Furthermore, we will write $x \neq y$ instead of $\neg(x = y)$ and $S^m(t)$ for the m-fold application of S to t, so $S^0(t) \equiv t$ and $S^{m+1}(t) \equiv S(S^m(t))$. Note that each $t \in W$ is of the form $S^m(u)$, for some $m \in \mathbb{N}$ and $u \in \bar{V}$.

We will use a fixed interpretation of the above formulas throughout this chapter. Terms are interpreted over the natural numbers ($\mathbb{N}$) and for formulas we use the classical interpretation over $\{0, 1\}$. Given a valuation $v : V \to \mathbb{N}$, we extend v homomorphically to terms and formulas in the following way:

$$\begin{array}{rcl} v(0) & = & 0 \\ v(S(t)) & = & 1 + v(t) \\ v(\bot) & = & 0 \\ v(\top) & = & 1 \\ v(s = t) & = & 1, \text{ if } v(s) = v(t), \text{ 0, otherwise.} \\ v(\neg\varphi) & = & 1 - v(\varphi) \\ v(\varphi \wedge \psi) & = & \min(v(\varphi), v(\psi)) \\ v(ITE(\varphi, \psi, \chi)) & = & v(\psi) \text{ if } v(\varphi) = 1, \ v(\chi) \text{ otherwise.} \end{array}$$

It is trivial that the valuation of a formula φ is 0 or 1.

Definition 2.3 *Given a formula φ, we say it is* satisfiable *if there exists a valuation $v : V \to \mathbb{N}$ such that $v(\varphi) = 1$; it is a* contradiction *otherwise. If for all $v : V \to \mathbb{N}$, $v(\varphi) = 1$, then φ is a* tautology*. Finally, if $v(\varphi) = v(\psi)$ for all valuations $v : V \to \mathbb{N}$, then φ and ψ are called* equivalent*.*

Lemma 2.4 *Every formula in Φ is equivalent to at least one $(0, S, =)$-BDD.*

Proof. First, we can eliminate all ITE symbols if we use the fact that $ITE(\varphi, \psi, \chi)$ and $\neg(\neg(\varphi \wedge \psi) \wedge \neg(\neg\varphi \wedge \chi))$ are equivalent. We prove the lemma by induction over the remaining formulas. $ITE(g, \top, \bot)$ is a suitable representation of a formula g when it is a guard. Now suppose φ_1, φ_2 are two given formulas with representations T_1, T_2, respectively. Construct a first $(0, S, =)$-BDD from T_1 by substituting T_2 for its $\top$ symbols and call it T. Construct a second $(0, S, =)$-BDD from T_1 by swapping $\top$ and $\bot$ in T_1 and name it T'. Now T and T' represent $\varphi_1 \wedge \varphi_2$ and $\neg\varphi_1$, respectively. ■

2.4 Elimination-Ordered $(0, S, =)$-BDDs

We now introduce a total ordering on guards. It will be used in the definition of *Elimination-Ordered* Binary Decision Diagrams ($(0, S, =)$-E-OBDDs). Next we prove that all $(0, S, =)$-BDDs (and hence all formulas) can be transformed to $(0, S, =)$-E-OBDDs by rewriting. Finally, we show that all paths in $(0, S, =)$-E-OBDDs represent consistent conjunctions, which make them well-suited for deciding satisfiability and contradiction of propositional formulas over zero, successor and equality.

2.4.1 Definition of $(0, S, =)$-E-OBDDs

From now on, BDD is an abbreviation to denote $(0, S, =)$-BDD. In this section, we define the set of *Elimination-Ordered* BDDs. To this end an ordering on guards is

needed. The latter is parameterized by a total ordering on the variables. In the sequel, we consider a fixed total and well-founded order on V. In the example below we assume that the variables x, y and z are ordered as $x \prec y \prec z$.

Definition 2.5 (ordering definition) *We extend $\prec$ to a total order on W:*

- $0 \prec u$ *for each element u of V*
- $S^m(x) \prec S^n(y)$ *if and only if $x \prec y$ or ($x \equiv y$ and $m < n$) for each two elements $x, y \in \bar{V}$*

As an illustration, the ordering above leads us to such a priority on terms:

$$x \prec S(x) \prec S^2(x) \prec \ldots \prec y \prec S(y) \prec S^2(y) \prec \ldots \prec z \prec S(z) \prec S^2(z)$$

We use term rewriting systems (TRS), being collections of rewrite rules, in order to specify reductions on guards and BDDs. The reduction relation induced by such a system is the closure of the rules under substitution and context. See [BN98] for a formal definition. A *normal form* is a term to which no rule applies. A TRS is *terminating* if all its reduction sequences are finite.

The first step to make a BDD ordered, is to simplify all its guards in isolation. Simplification on guards is defined by the following rules.

Definition 2.6 *Suppose g is a guard. By $g\downarrow$ we mean the normal form of g w.r.t. the following rewrite rules:*

$$\begin{aligned} x = x &\rightarrow \top \\ S(y) = S(x) &\rightarrow y = x \\ S(x) = 0 &\rightarrow \bot \\ S^{m+1}(x) = x &\rightarrow \bot && \textit{for all } m \in \mathbb{N} \\ r = t &\rightarrow t = r && \textit{for all } r, t \in W \textit{ such that } r \prec t \end{aligned}$$

We call g simplified if it cannot be further simplified, i.e. $g \equiv g\downarrow$. The last rule above, places the bigger term of the guard, on its left side. Hence for instance $S^m(y) = x$ does not change by this rule if $x \prec y$.

Remark 2.7 *Suppose $g \in G$ is a guard which becomes g' after applying a certain rule of Definition 2.6 on it. Then g and g' are equivalent, i.e. they will have the same value under each valuation function.*

Proof. We show this for the fourth rule. Suppose $v : V \rightarrow \mathbb{N}$ is an arbitrary valuation function which is homomorphically extended to terms and formulas. Now we compare the value of v on the two sides of the fourth rule: $v(S^{m+1}(x) = x) = 1$ if $v(S^{m+1}(x)) = v(x)$ otherwise $v(S^{m+1}(x) = x) = 0$. Obviously $v(S^{m+1}(x)) = m + 1 + v(x) \neq v(x)$.

Hence $v(S^{m+1}(x) = x) = 0$. On the other hand $v(\bot) = 0$. Therefore these two are equivalent. Other cases are similar.

■

Definition 2.8 (Simplified BDD) *A $(0, S, =)$-BDD T is called simplified if all its guards are simplified.*

An immediate consequence of Definition 2.6 is the following:

Corollary 2.9 *Each simplified guard has exactly one of the following forms:*

- $x = S^m(0)$ *for some* $x \in V$, $m \geq 0$
- $y = S^m(x)$ *for some* $x, y \in V$, $x \prec y$, $m > 0$
- $S^m(y) = x$ *for some* $x, y \in V$, $x \prec y$, $m > 0$
- $\top$, $\bot$

Below we first illustrate the kind of question we need to deal with in order to be able later on to obtain a serious of reducing BDDs which will after finite steps, lead us to a halting point where it cannot be further reduced.

Hint. Consider the following formula: $\varphi := (S^3(y) = x) \wedge (s(y) = x)$. Since it is unsatisfiable, $\bot$ must become the only OBDD which represents φ. So the question is how we can obtain $\bot$ from a BDD representation of φ in a systematic way. The first answer which comes to mind might be the replacement of x in $S^3(y) = x$ by $S(y)$. But this will produce a bigger term (with respect to $\prec$) in the formula, and in a general case if we are not lucky enough to have it simplified (which in this example is possible) then this kind of replacements will cause a growth and a blow-up in the size of the formula. Therefore finding a proper replacement for terms like these is very important. It should be in a way that it can be used in general, and will reduce the size of BDDs. Here we solve it by a lifting process which raises the second equation by $S^2(.)$ to obtain $S^3(y) = S^2(x)$, then substitute $S^3(y)$ by x which is the right-hand side of the first equality. Result can be simplified to $\bot$ (by Definition 2.6). Next example is also another demonstration for how we deal with these replacements.

Example 2.10 *Let $x \prec y \prec z$ (see Figure 2.2).*

$$
\begin{aligned}
& ITE(S^2(y) = x, ITE(z = y, \top, \bot), \bot) \\
\stackrel{*}{\rightarrow}\ & ITE(S^2(y) = x, ITE(\{(S^2(z) = S^2(y))[S^2(y) := x]\}\downarrow, \top, \bot), \bot) \\
\equiv\ & ITE(S^2(y) = x, ITE(S^2(z) = x, \top, \bot), \bot)
\end{aligned}
$$

here * indicates the kind of rule which we want to have. First it applies the function S two times on z and y. Then in the resulting guard it replaces $S^2(y)$ with x. Later on we will explain that the resulting BDD is smaller this way.

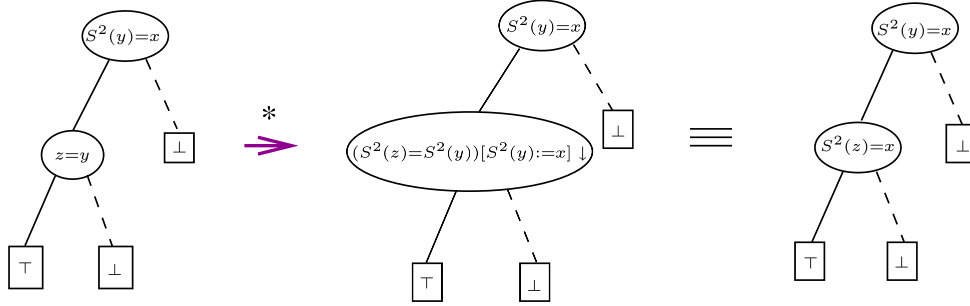

Figure 2.2: Derivation in Example 2.10

Lifting (with function S) and substitution are defined below, and we will show later that in combination with simplification, these operations result in smaller guards.

Definition 2.11 *Let $m \in \mathbb{N}$, terms $r, t \in W$, a variable $y \in V$ and a guard $g \in G$ be given. Then we define:*

$$(r = t){\uparrow}^m := S^m(r) = S^m(t) \qquad \text{(lifting)}$$
$$g|_{S^m(y)=r} := (g{\uparrow}^m\ [S^m(y) := r])\downarrow \qquad \text{(substitution)}$$

Here $S^m(y) := r$ denotes the replacement of the term $S^m(y)$ by r.

The following remark shows that the operations above our sound:

Remark 2.12 *For any guard g and a positive natural number m, $g{\uparrow}^m$ and g are equivalent terms. Moreover suppose f is another guard. If f holds under a valuation v then g and $g|_f$ will be equivalent under v.*

Proof. The first part is trivial, since each valuation will have the same value on both $g{\uparrow}^m$ and g. Now suppose f is of the form $S^m(z) = y$ and v satisfies f. Hence $v(S^m(z)) = v(y)$. Let g be $t = z$, and hence $g|_f$ is $S^m(t) = y$. Then

$$\begin{aligned} v(t = z) = 1 &\Leftrightarrow v(t) = v(z) \\ &\Leftrightarrow v(S^m(t)) = v(S^m(z)) \\ &\Leftrightarrow v(S^m(t)) = v(y) \\ &\Leftrightarrow v(S^m(t) = y) = 1. \end{aligned}$$

■

As it was mentioned before, to impose an ordering on BDDs, first we need a total ordering on guards. Since we are going to deal with simplified guards, we limit our definition to the simplified guards.

Definition 2.13 (order on simplified guards) *We define a total order $\prec$ on simplified guards as below*

- $\bot \prec \top \prec g$, *for all guards* g, *different from* $\top$, $\bot$.
- *for* $x, y \in V$: $(S^p(x) = S^q(y)) \prec (S^m(u) = S^n(v))$ *iff*

$$\begin{aligned} i)\ & x \prec u \text{ or} \\ ii)\ & x \equiv u,\ y \prec v \text{ or} \\ iii)\ & x \equiv u,\ y \equiv v,\ p < m \text{ or} \\ iv)\ & x \equiv u,\ y \equiv v,\ p = m,\ q < n \end{aligned}$$

This order is well-defined, because it is applied on simplified guards. According to this definition $S^p(x) = S^q(y) \prec S^m(u) = S^n(v)$ iff $(x, y, p, q) \prec_{lex} (u, v, m, n)$, in which $\prec_{lex}$ is a lexicographic order on quadruples of the total, well-founded orders $(\bar{V}, \prec) \times (\bar{V}, \prec) \times (\mathbb{N}, <) \times (\mathbb{N}, <)$, and therefore it is well-founded and total.

Now we have all we need in order to start the procedure of transforming $(0, S, =)$-BDDs to equivalent $(0, S, =)$-E-OBDDs. Below a set of rules, called a *Term Rewrite System* (TRS) are introduced. By applying this TRS on any given simplified $(0, S, =)$-BDD we will obtain an equivalent $(0, S, =)$-E-OBDD.

As it is described in [GvdP00], Ordered EQ-BDDs are not necessarily unique, so for our extension, we will not make any attempt to obtain unique representations. Next definition presents a system which operates on simplified BDDs.

Definition 2.14 ($(0, S, =)$-E-OBDD) *An $(0, S, =)$-E-OBDD (Elimination-Ordered $(0, S, =)$-BDD) is a $(0, S, =)$-BDD which is simplified and is a normal form w.r.t. to the following rewrite rules (these rules are applied only on simplified BDDs):*

1. $ITE(\top, T_1, T_2) \to T_1$
2. $ITE(\bot, T_1, T_2) \to T_2$
3. $ITE(g, T, T) \to T$
4. $ITE(g, ITE(g, T_1, T_2), T_3) \to ITE(g, T_1, T_3)$
5. $ITE(g, T_1, ITE(g, T_2, T_3)) \to ITE(g, T_1, T_3)$
6. $ITE(g_1, ITE(g_2, T_1, T_2), T_3) \to ITE(g_2, ITE(g_1, T_1, T_3), ITE(g_1, T_2, T_3))$ *if* $g_1 \succ g_2$
7. $ITE(g_1, T_1, ITE(g_2, T_2, T_3)) \to ITE(g_2, ITE(g_1, T_1, T_2), ITE(g_1, T_1, T_3))$ *if* $g_1 \succ g_2$

8. *For any simplified* $(0, S, =)$*-BDD* C*,* $g \in G$*,* $r \in W$*,* $y \in V$ *and* $m \in \mathbb{N}$*:*
   $ITE(S^m(y) = r, C[g], T) \rightarrow ITE(S^m(y) = r, C[g|_{S^m(y)=r}], T)$
   if y *occurs in* g *and* $S^m(y) = r \prec g$

Here $C[g]$ represents a BDD C as a context including a hole which is filled in with guard g. $C[g|_{S^m(y)=r}]$ denotes the replacement of $S^m(y)$ by $S^m(y) = r$ in any occurrence of g in C. It is obvious that the result of applying any rule on a simplified BDD is again a simplified BDD. Rules 1–7 are the normal rules for simplifying BDDs for plain propositional logic [ZvdP01], which remove redundant tests, and ensure that guards along paths occur in increasing order (Lemma 2.16). Rule 8 allows to substitute equals for equals. Example 2.10 is a demonstration for this rule, there * expresses rule 8 (above) (see **Hint** before Example 2.10). This rule is needed to take care of transitivity of equality. Other properties of equality, such as reflexivity, symmetry, and injectivity of successor, are dealt with by the simplification rules (see Example 2.6 and Remark 2.7). The reason we call the outcome Elimination-Ordered $(0, S, =)$-BDD is that according to rule 8 the variable occurring in the left-hand side of a guard is "eliminated" from the underneath guards. This variable also has the first priority to determine where the atom $S^m(y) = r$ will sit in the ordering process. So the ordering is in a way according to the variable which is eliminated. In the remainder of this chapter we frequently talk about OBDDs instead of $(0, S, =)$-E-OBDDs.

Remark 2.15 *Suppose* $T \in B$ *is a* $(0, S, =)$*-BDD which becomes* T' *after applying an arbitrary rule of Definition 2.14 on it. Then* T *and* T' *are equivalent, i.e. they will have the same value under each valuation function. As a result each* $(0, S, =)$*-BDD is equivalent with its normal form (out of Definition 2.14).*

Proof. We show this for rule 6 and rule 8. Suppose $v : V \rightarrow \mathbb{N}$ is an arbitrary valuation function which is homomorphically extended to terms and formulas. Now we compare the value of v on the two sides of rule 6:

$$v(ITE(g_1, ITE(g_2, T_1, T_2), T_3)) := \begin{cases} v(T_1) & \text{if } v(g_1) = v(g_2) = 1 \\ v(T_2) & \text{if } v(g_1) = 1 \text{ and } v(g_2) = 0 \\ v(T_3) & \text{otherwise} \end{cases}$$

On the other hand

$$v(ITE(g_2, ITE(g_1, T_1, T_2), ITE(g_1, T_1, T_3)) := \begin{cases} v(T_1) & \text{if } v(g_1) = v(g_2) = 1 \\ v(T_2) & \text{if } v(g_1) = 1,\ v(g_2) = 0 \\ v(T_3) & \text{otherwise} \end{cases}$$

Function v is equal for both in all the conditions, hence these two are equivalent. With a similar calculation for rule 8 we have:

$$v(ITE(S^m(y) = r, C[g], T)) := \begin{cases} v(C[g]) & \text{if } v(S^m(y) = r) = 1 \\ v(T) & \text{otherwise} \end{cases}$$

On the other hand

$$v(ITE(S^m(y) = r, C[g|_{S^m(y)=r}], T)) := \begin{cases} v(C[g|_{S^m(y)=r}]) & \text{if } v(S^m(y) = r) = 1 \\ v(T) & \text{otherwise} \end{cases}$$

These two are equivalent according to Remark 2.12. The equivalence of the two sides of other rules can similarly be proved. ∎

Next lemma visualizes rules 6 and 7:

Lemma 2.16 *Let T be an OBDD. Suppose g_1 and g_2 occur on a path α in T and $g_1 \succ g_2$, then g_1 is placed below g_2 on α.*

Proof. It is trivial with respect to the rules 2.14.6 and 2.14.7. ∎

We now show with an example how a $(0, S, =)$-BDD becomes an $(0, S, =)$-E-OBDD:

Example 2.17 *Let $x \prec y \prec z$ (see Figure 2.3).*

$$\begin{array}{cl} & ITE(z = S(y), ITE(S^2(y) = x, \top, \bot), \bot) \\ \xrightarrow{6} & ITE(S^2(y) = x, ITE(z = S(y), \top, \bot), ITE(z = S(y), \bot, \bot)) \\ \xrightarrow{3} & ITE(S^2(y) = x, ITE(z = S(y), \top, \bot), \bot) \\ \xrightarrow{8} & ITE(S^2(y) = x, ITE(\{S^2(z) = S^3(y)[S^2(y) := x]\}\downarrow, \top, \bot), \bot) \\ \overset{substitution}{\equiv} & ITE(S^2(y) = x, ITE(\{S^2(z) = S(x)\}\downarrow, \top, \bot), \bot) \\ \equiv & ITE(S^2(y) = x, ITE(S(z) = x, \top, \bot), \bot) \end{array}$$

2.4.2 Termination

Now we present the first main claim that every BDD (with zero, successor and equality) has a normal form with respect to the rewrite system of Definition 2.14, which implies that each BDD has at least one equivalent OBDD. It suffices to prove termination. We can then apply TRS rules to a given BDD, until a normal form is reached after a finite number of steps, which is guaranteed by termination. The so derived BDD is an equivalent OBDD.

We prove termination by means of *recursive path order* ($\prec_{rpo}$) [BN98, Der87]. The main idea behind recursive path order is that two formulas are compared first by

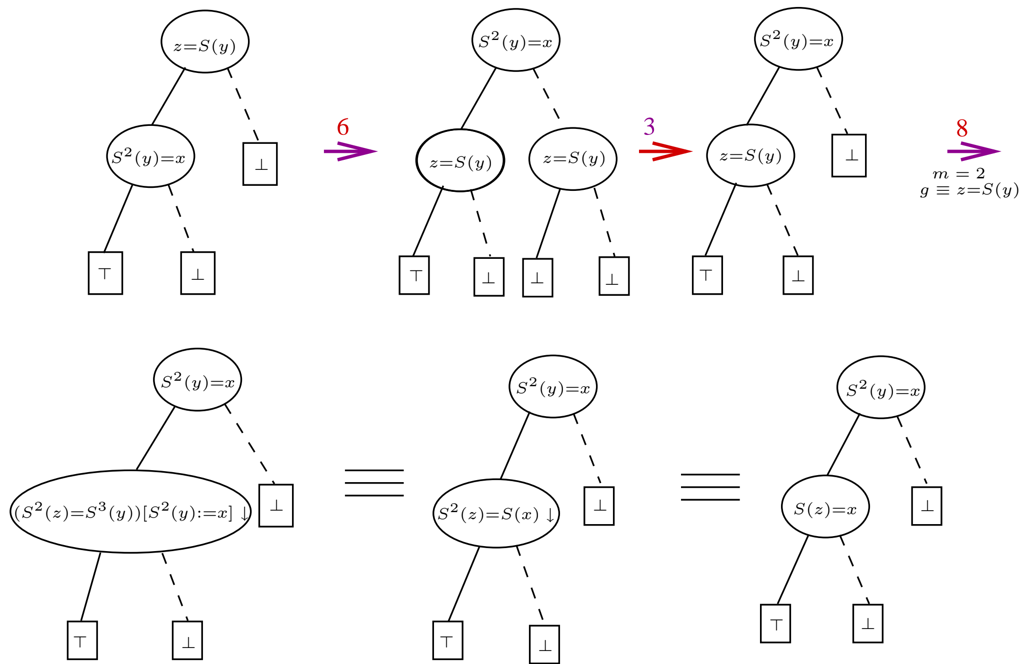

Figure 2.3: Derivation in Example 2.17

comparing their root (here it will be the leading guard) and then recursively comparing their immediate subformulas. This is a standard way to extend a (total) well-founded order on a set of labels to a (total) well-founded order on trees (i.e. a kind of term) over these labels. To this end, we view guards as labels, ordered by Definition 2.13, and BDDs are viewed as *binary* trees, so $ITE(g, T_1, T_2)$ corresponds to the tree $g(T_1, T_2)$.

Definition 2.18 (recursive path order for BDDs) *Let S and T be simplified BDDs. Then $S \equiv f(S_1, S_2) \succ_{rpo} g(T_1, T_2) \equiv T$ if and only if*

(I) $S_1 \succeq_{rpo} T$ *or* $S_2 \succeq_{rpo} T$*; or*

(II) $f \succ g$ *and* $S \succ_{rpo} T_1, T_2$*; or*

(III) $f \equiv g$ *and* $S \succ_{rpo} T_1, T_2$ *and either* $S_1 \succ_{rpo} T_1$*, or* ($S_1 \equiv T_1$ *and* $S_2 \succ_{rpo} T_2$)*.*

Here $x \succeq_{rpo} y$ means that $x \succ_{rpo} y$ or $x \equiv y$, and $S \succ_{rpo} T_1, T_2$ is shorthand for $S \succ_{rpo} T_1$ and $S \succ_{rpo} T_2$.

This definition yields an order, as is shown in [Zan03]. The next remark mentions that if a sub-tree of a BDD is replaced by a smaller tree, then the whole tree will get smaller.

Remark 2.19 *Let T be a simplified BDD, and S be a sub BDD of it. Suppose S' is a new simplified BDD. We replace S by S' in T. The new BDD is called T'. If $S \succ_{rpo} S'$ then $T \succ_{rpo} T'$*

Proof. It is easy by induction on the structure of T and Definition 2.18(III). ■

In order to prove termination, we will show that each rewrite rule (of Definition 2.14) is indeed a reduction rule regarding $\succ_{rpo}$. The next lemma will be very helpful to show that this reduction property really holds.

Lemma 2.20 *Let $f \equiv S^n(y) = S^m(x)$ and $g \equiv S^k(w) = S^l(v)$ be simplified guards. If $f \prec g$ and $f \equiv f\downarrow$ and $g \equiv g\downarrow$ and $y \in \{v, w\}$, then $g|_f \prec g$.*

Proof.

- Case I: $y \equiv v$. Hence $x \prec y \equiv v \prec w$, since f and g are simplified guards. Now

$$\begin{aligned} g|_f &\equiv (g{\uparrow}^n\ [S^n(y) := S^m(x)])\downarrow \\ &\equiv (S^{k+n}(w) = S^{l+m}(x))\downarrow && v \equiv y \\ &\prec S^k(w) = S^l(v) && x \prec v,\ \text{Definition 2.13(ii)} \\ &\equiv g \end{aligned}$$

- Case II: $y \equiv w$. Hence $y \equiv w \succ v$, since g is a simplified guard. Now

$$
\begin{aligned}
g|_f &\equiv (g{\uparrow}^n\ [S^n(y) := S^m(x)])\!\downarrow \\
&\equiv (S^{k+m}(x) = S^{l+n}(v))\!\downarrow \qquad\qquad w \equiv y
\end{aligned}
$$

By Definition 2.13(i), $(S^{k+m}(x) = S^{l+n}(v))\!\downarrow \prec S^k(y) = S^l(v)$, irrespective of whether $x \prec v$ or $v \prec x$, because $x \prec y$ and $v \prec y$.

■

Lemma 2.21 *Let f, g be two simplified guards, such that $f \prec g$, and C is a $(0, S, =)$-BDD. If g occurs at least once in C, then $C[g] \succ_{rpo} C[f]$.*

Proof. Monotonicity of $\succ_{rpo}$ [Zan03]. ■

Lemma 2.22 *All rewrite rules are contained in $\succ_{rpo}$, i.e. after applying a rule on a BDD, the result is smaller than the input BDD.*

Proof.

1. $\top(T_1, T_2) \succ_{rpo} T_1$ by (I).
2. Similarly.
3. $g(T, T) \succ_{rpo} T$ by (I).
4. $g(g(T_1, T_2), T_3) \succ_{rpo} g(T_1, T_3)$ by (III) and (I).
5. Similarly.
6. Assume $g_1 \succ g_2$ and let $S \equiv g_1(g_2(T_1, T_2), T_3)$. Then
   - $S \succ_{rpo} T_3$ by (I)
   - $g_2(T_1, T_2) \succ_{rpo} T_1$ by (I)
   - $S \succ_{rpo} T_1$ by (I)

   hence $S \succ_{rpo} g_1(T_1, T_3)$ by (III). Similarly $S \succ_{rpo} g_1(T_2, T_3)$. And therefore $S \succ_{rpo} g_2(g_1(T_1, T_3), g_1(T_2, T_3))$ by (II).
7. Similarly.
8. Let $f \equiv S^m(y) = S^n(x)$. Assume y occurs in g and $f \prec g$, and f and g are simplified. We have to show that $f(C[g], T) \succ_{rpo} f(C[\ g|_f\ ], T)$. Using Lemma 2.20 we conclude $g \succ g|_f$, and so $C[g] \succ_{rpo} C[g|_f]$ by Lemma 2.21. Now, by using (I) twice and next (III), it is clear that this rule is also contained in $\succ_{rpo}$.

■

Now we are able to prove our first main claim:

Theorem 2.23 *The rewrite system in Definition 2.14 is terminating on simplified* $(0, S, =)$*-BDDs.*

Proof. We showed in the previous lemma that all rewrite rules are contained in $\succ_{rpo}$. This implies termination, because $\succ_{rpo}$ is a reduction order, i.e. well-founded, and closed under substitutions and contexts [Zan03]. ■

This theorem says that by repeated applications of the rewrite rules on an arbitrary simplified BDD, after finitely many iterations we will obtain the normal form of it, which is its equivalent ordered form, so

Corollary 2.24 *Every* $(0, S, =)$*-BDD is equivalent to at least one* $(0, S, =)$*-E-OBDD.*

Proof. According to Theorem 2.23, every $(0, S, =)$-BDD becomes a $(0, S, =)$-E-OBDD using the rewrite system of Definition 2.14. By Remark 2.15 these two are equivalent. Hence, every $(0, S, =)$-BDD is equivalent to at least one $(0, S, =)$-E-OBDD. ■

2.4.3 Satisfiability of paths in $(0, S, =)$-E-OBDDs

For a given formula, we can now construct a BDD representation (Lemma 2.4) and turn it to an OBDD by rewriting (Corollary 2.24). We now show the second main claim, stating that all paths in an ordered BDD represent satisfiable conjunctions. As a consequence it can be decided whether the formula is a tautology, a contradiction or it is satisfiable.

NOTATION. Let α, β, γ range over finite sequences of guards and negations of guards. We write ε for the empty sequence, and $\alpha.\beta$ for the concatenation of sequences α and β. If the order of a sequence is unimportant, we sometimes view it as a set, and write $g \in \alpha$, or even $\alpha \cup \beta$. The latter denotes the set of all guards or negations of guards that occur somewhere on α or β.

Definition 2.25 Literals *are guards or negations of guards.* Paths *are sequences of literals. We define the set of paths of a* $(0, S, =)$*-BDD:*

- $Pat(\top) = Pat(\bot) = \{\varepsilon\}$
- $Pat(ITE(g, T_1, T_2)) = \{g.\alpha \mid \alpha \in Pat(T_1)\} \cup \{\neg g.\beta \mid \beta \in Pat(T_2)\}$

A path α *is* ordered *if it is a path in some* $(0, S, =)$*-E-OBDD. Valuation* $v : V \to \mathbb{N}$ satisfies α *if* $v(g) = 1$ *for all literals* $g \in \alpha$*.* α *is* satisfiable *if a valuation* v *that satisfies it exists.*

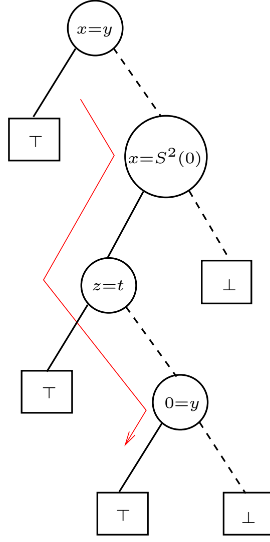

Figure 2.4: A path from Example 2.27

Definition 2.26 *Let α be a path in a given BDD. We say α ends in $\top$ (resp. $\bot$) if the conjunction of its elements (guards) makes the formula representation of the BDD $\top$ (resp. $\bot$).*

Example 2.27 *Let*

$$T \equiv ITE(x = y, \top, ITE(x = S^2(0), ITE(z = t, \top, ITE(0 = y, \top, \bot)), \bot)).$$

Then $x \neq y$. $x = S^2(0)$. $z \neq t$. $0 = y$ is a path (Figure 2.4). This path ends in $\top$. Furthermore, let $x \prec y \prec z$. Then $y = x$. $z = x$ is an ordered path, because it is a path in $ITE(y = x, ITE(z = x, \top, \bot), \bot)$, which is an OBDD.

The following two lemmas give some syntactical properties on OBDDs, which can be used for proving satisfiability of each path in an OBDD.

Lemma 2.28 *Let α be an ordered path, of the form $\beta.(S^p(u) = S^q(y)).\gamma$. Then:*

(i) u does not occur in γ

(ii) u does not occur at the right-hand side of any literal in β

(iii) u does not occur in a positive guard in β

(iv) y does not occur at the left-hand side of any literal in γ

Proof.

- (i) Since α is ordered, the rewrite rules should not be applicable. If u occurs in $g \in \gamma$, then either $S^p(u) = S^q(y) \prec g$, and hence rule 8 is applicable, or $S^p(u) = S^q(y) \succeq g$, and one of rules 4-7 is applicable.

- (ii) Because otherwise, if $g \equiv S^k(v) = S^l(u)$ occurs in β, then $v \succ u$, so $g \succ S^p(u) = S^q(y)$, which contradicts the fact that α is ordered.

- (iii) Regarding part (ii) above, u can occur only in the left-hand side of a positive guard like $S^i(u) = S^j(z)$ in β. In that case two paths β' and γ' exist, such that $\alpha \equiv \beta'.(S^i(u) = S^j(z)).\gamma'$, and $S^p(u) = S^q(y)$ belongs to γ'. But referring to part (i), this will never happen.

- (iv) For a similar reason as part (ii).

■

Lemma 2.29 *Suppose $S^l(u) = S^k(y)$ and $S^p(u) \neq S^q(y)$ are two literals on an ordered path δ. If v is a valuation which satisfies $S^l(u) = S^k(y)$, then it also satisfies $S^p(u) \neq S^q(y)$.*

Proof.

- If $S^l(u) = S^k(y) \prec S^p(u) = S^q(y)$, then two paths β and γ exist such that $\delta \equiv \beta.(S^l(u) = S^k(y)).\gamma$ in which $S^p(u) \neq S^q(y)$ belongs to γ, but according to Lemma 2.28(i), this will never happen.

- If $S^p(u) = S^q(y) \prec S^l(u) = S^k(y)$, then since δ is ordered, we can limit our inquiry to the two following cases:

  - $p < l$, and so $k = 0$:

$$
\begin{aligned}
v(S^p(u)) &= p + v(u) \\
&< l + v(u) \\
&= k + v(y) && v \text{ satisfies } S^l(u) = S^k(y) \\
&= v(y) && k = 0 \\
&\leq q + v(y) \\
&= v(S^q(y))
\end{aligned}
$$

– $p = l$ and $q < k$:

$$
\begin{aligned}
v(S^p(u)) &= p + v(u) \\
&= l + v(u) && p = l \\
&= k + v(y) && v \text{ satisfies } S^l(u) = S^k(y) \\
&> q + v(y) && q < k \\
&= v(S^q(y))
\end{aligned}
$$

In both of these two cases, $v(S^p(u)) \neq v(S^q(y))$. ■

Definition 2.30 *Suppose $s = t$ is a guard and α is a path. Define:*

$$
\begin{aligned}
&\mathit{Reverse}(s = t) := t = s \\
&\bar{\alpha} := \alpha \cup \{\ \mathit{Reverse}(g) \mid g \in\ \alpha\} \cup \{\ \neg\mathit{Reverse}(g) \mid \neg g \in\ \alpha\}
\end{aligned}
$$

Definition 2.31 *Suppose α is an ordered path of the form $\beta.(S^m(z) = S^n(x)).\gamma$. We define a set $E_{x\alpha}$ as follows:*

$$E_{x\alpha} = \{u \in \bar{V} \mid S^p(u) = S^q(x) \in \alpha \text{ for some } p, q \in \mathbb{N}\}$$

Remark 2.32 *According to Definition 2.31, 0 does not belong to $E_{x\alpha}$, because $S^p(u) = S^q(x)$ is a simplified guard on the ordered path α. Therefore $u \succ x$, but we know that 0 does not have this property.*

Intuitively, the set $E_{x\alpha}$ contains all variables from terms that are related to x by path α. So if we want to raise the value of x, we must raise all values in $E_{x\alpha}$ as well. Note that the value of 0 can not be raised, and raising the value of x could inadvertently make some negated guards in α true. These considerations are captured by the following lemma, which shows how a given valuation that satisfies α can be lifted to arbitrarily high values.

Lemma 2.33 *Given α, an ordered path of the form $\beta.(S^m(z) = S^n(x)).\gamma$, in which $x \in V$ (i.e. $x \not\equiv 0$), and given a valuation v which satisfies this path. Then for each $k \in \mathbb{N}$ exists $l > k$ and a valuation v', such that*

$$
\begin{aligned}
&(i)\ v' \text{ satisfies } \alpha \\
&(ii)\ v'(u) = v(u) + l && \text{for each } u \in E_{x\alpha} \cup \{x\} \\
&(iii)\ v'(y) = v(y) && \text{for each } y \notin E_{x\alpha} \cup \{x\}
\end{aligned}
$$

Proof. Let us give some notes, before defining any valuation v'.

Note 1. If $S^p(u) = S^q(y)$ is a positive guard on α and $y \not\equiv x$, then u does not belong to $E_{x\alpha} \cup \{x\}$.

Proof.

- $u \not\equiv x$, because otherwise α would be of the form $\mu.(S^p(x) = S^q(y)).\delta$ for some ordered paths μ and δ, and $S^m(z) = S^n(x) \in \mu \cup \delta$. If it is in μ, this contradicts Lemma 2.28(iii). If it is in δ, this contradicts Lemma 2.28(i).

- $u \notin E_{x\alpha}$, because otherwise $S^i(u) = S^j(x) \in \alpha$, for some $i, j \in \mathbb{N}$, and this guard will be different from $S^p(u) = S^q(y)$, since $y \not\equiv x$. Therefore $S^i(u) = S^j(x) \prec S^p(u) = S^q(y)$ or vice versa. In each of these two cases, a contradiction is derived, regarding Lemma 2.28(i). □

Note 2. If for some $u \in \bar{V}$ and $p, q \in \mathbb{N}$, $S^p(u) = S^q(y)$ occurs positively or negatively in α, then y does not belong to $E_{x\alpha}$.

Proof. If $y \in E_{x\alpha}$ then $S^i(y) = S^j(x) \in \alpha$ for some $i, j \in \mathbb{N}$. $y \prec u$, since $S^p(u) = S^q(y)$ is a simplified guard on the ordered path α, therefore $S^i(y) = S^j(x) \prec S^p(u) = S^q(y)$. This means that the ordered path $\alpha \equiv \mu.(S^i(y) = S^j(x)).\delta$ for some μ and δ, in which $S^p(u) = S^q(y)$ or its negation belongs to δ. But this contradicts Lemma 2.28(i). □

Now define:

$$m' = max\{q + v(y) \mid y \neq x \text{ and } \exists u \in \{x\} \cup E_{x\alpha},\ \exists j \in \mathbb{N} :\ S^j(u) \neq S^q(y) \in \bar{\alpha}\}$$

Intuitively, m' is greater than everything distinct from $E_{x\alpha}$. Using this m', we introduce a new valuation v' as below:

$$v'(u) := \begin{cases} v(u) + m' + k + 1 & \text{if } u \in \{x\} \cup E_{x\alpha} \\ v(u) & \text{otherwise} \end{cases}$$

$x \not\equiv 0$ by the assumption. Moreover, given $u \in E_{x\alpha}$, u is nonzero by Remark 2.32. Therefore the given definition for v' is well-defined. Now define $l := m' + k + 1$. Then requirements (ii) and (iii) of the lemma are obviously met. Below we will show that requirement (i) holds, i.e. v' satisfies α. Suppose g is a literal on this path.

- If $g \equiv S^p(u) = S^q(y)$, then either of the two following cases applies:

- $y \equiv x$. Then $u \in E_{x\alpha}$, and hence $v'(u) = v(u) + m' + k + 1$. Now:

$$
\begin{aligned}
v'(S^p(u)) &= p + v'(u) \\
&= p + v(u) + m' + k + 1 && v'(u) = v(u) + m' + k + 1 \\
&= v(S^p(u)) + m' + k + 1 \\
&= v(S^q(y)) + m' + k + 1 && v \text{ satisfies } \alpha \\
&= q + v(x) + m' + k + 1 && y \equiv x \\
&= q + v'(x) && v'(x) = v(x) + m' + k + 1 \\
&= v'(S^q(y)) && y \equiv x
\end{aligned}
$$

- $y \not\equiv x$. Now according to the two given notes, u and y both belong to the last case of the definition of v'. Therefore:

$$
\begin{aligned}
v'(S^p(u)) &= p + v'(u) \\
&= p + v(u) && v'(u) = v(u) \\
&= v(S^p(u)) \\
&= v(S^q(y)) && v \text{ satisfies } \alpha \\
&= q + v(y) \\
&= q + v'(y) && v'(y) = v(y) \\
&= v'(S^q(y))
\end{aligned}
$$

- If $g \equiv S^p(u) \neq S^q(y)$, then $y \notin E_{x\alpha}$, by Note 2. We distinguish two cases:
  - $u \in E_{x\alpha}$. Therefore:
    * If $y \equiv x$, then, since $u \in E_{x\alpha}$, $S^i(u) = S^j(x) \in \alpha$ for some $i, j \in \mathbb{N}$, and according to the previous case, $v'(S^i(u)) = v'(S^j(x))$. Hence $v'(S^p(u)) \neq v'(S^q(x))$ by Lemma 2.29.
    * If $y \not\equiv x$, then $v'(y) = v(y)$ because y also does not belong to $E_{x\alpha}$. Hence:

$$
\begin{aligned}
v'(S^p(u)) &= v'(u) + p \\
&= v(u) + m' + k + 1 + p && u \in E_{x\alpha} \\
&= v(S^p(u)) + m' + k + 1 \\
&> m' \\
&\geq v(S^q(y)) && \text{definition of } m' \\
&= v'(S^q(y)) && v'(y) = v(y)
\end{aligned}
$$

  - $u \notin E_{x\alpha}$. Thus:

* If $u \equiv x$, then $y \not\equiv x$, since g is simplified. So y belongs to the last case of the definition of v', because $y \notin E_{x\alpha}$, and hence $v'(y) = v(y)$. Now:

$$\begin{aligned} v'(S^p(u)) &= v'(S^p(x)) && u \equiv x \\ &= v'(x) + p \\ &= v(S^p(x)) + m' + k + 1 \\ &> m' \\ &\geq v(S^q(y)) && u \equiv x, \text{ definition on } m' \\ &= v'(S^q(y)) && v'(y) = v(y) \end{aligned}$$

* If $u \not\equiv x$, then $v'(u) = v(u)$, since $u \notin E_{x\alpha}$. Therefore:
  · If $y \equiv x$, then

$$\begin{aligned} v'(S^p(u)) &= v'(u) + p \\ &= v(u) + p \\ &\leq m' && y \equiv x, \text{ definition of } m' \\ &< v(x) + m' + k + 1 \\ &= v'(x) \\ &\leq v'(S^q(x)) \\ &= v'(S^q(y)) && y \equiv x \end{aligned}$$

  · If $y \not\equiv x$, then y also belongs to the last case of the definition of v', because $y \notin E_{x\alpha}$. Thus:

$$\begin{aligned} v'(S^p(u)) &= v'(u) + p \\ &= v(u) + p \\ &= v(S^p(u)) \\ &\neq v(S^q(y)) && v \text{ satisfies } \alpha,\ S^p(u) \neq S^q(y) \in \alpha \\ &= q + v(y) \\ &= q + v'(y) \\ &= v'(S^q(y)). \end{aligned}$$

■

Finally, we come to the second main claim of this chapter:

Theorem 2.34 *Each path in a $(0, S, =)$-E-OBDD is satisfiable.*

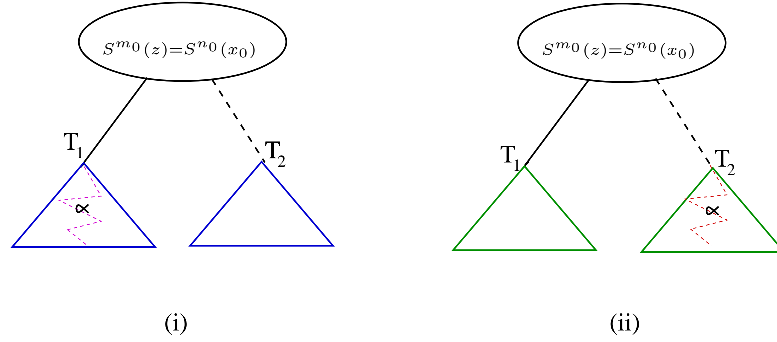

Figure 2.5: Satisfiable path α in OBDD T_1 or T_2

Proof. We prove this theorem by induction over OBDDs. Suppose $T \equiv ITE(S^{m_0}(z) = S^{n_0}(x_0), T_1, T_2)$ is an OBDD, and each path belonging to T_1 or T_2, is satisfiable. We will show that each path in T is satisfiable as well.
Consider α is a satisfiable path, and v is a valuation which satisfies it.

Supposing α belongs to T_1 (see Figure 2.5(i)), we provide a new valuation, which satisfies $(S^{m_0}(z) = S^{n_0}(x_0)).\alpha$. Since T is ordered, z does not occur in any literal of α, by Lemma 2.28(i).

- If $x_0 \equiv 0$: then $m_0 = 0$, since $S^{m_0}(z) = S^{n_0}(x_0)$ is a simplified guard (Corollary 2.9). Define:

$$v'(u) := \begin{cases} n_0 & \text{if } u \equiv z \\ v(u) & \text{otherwise} \end{cases}$$

  v' obviously satisfies $(S^{m_0}(z) = S^{n_0}(x_0)).\alpha$.

- If $x_0 \not\equiv 0$: z does not occur on α, so that $(z = x_0).\alpha$ is still an ordered path, and without loss of generality, we can define $v(z) := v(x_0)$. Therefore, v satisfies $(z = x_0).\alpha$. Using Lemma 2.33, there is a valuation v' and a natural number $l > m_0$ such that v' satisfies $(z = x_0).\alpha$ and $v'(x_0) = v(x_0) + l$. Now define:

$$v''(u) := \begin{cases} v'(x_0) + n_0 - m_0 & \text{if } u \equiv z \\ v'(u) & \text{otherwise} \end{cases}$$

v'', is well-defined since

$$\begin{aligned} v''(z) &= v'(x_0) + n_0 - m_0 \\ &= v(x_0) + l + n_0 - m_0 \\ &= v(x_0) + n_0 + (l - m_0) \\ &\geq 0. \end{aligned}$$

v'' satisfies α since v' does. Moreover

$$\begin{aligned} v''(S^{m_0}(z)) &= m_0 + v''(z) && \text{definition of } v''(z) \\ &= v'(x_0) + n_0 \\ &= v''(x_0) + n_0 \\ &= v''(S^{n_0}(x_0)) \end{aligned}$$

which means, v'' satisfies $S^{m_0}(z) = S^{n_0}(x_0)$. Therefore $(S^{m_0}(z) = S^{n_0}(x_0)).\alpha$ is satisfiable.

Supposing α belongs to T_2 (see Figure 2.5(ii)), we provide a new valuation, which satisfies $(S^{m_0}(z) \neq S^{n_0}(x_0)).\alpha$. Define:

$$H := \bar{\alpha} \cup \{S^{m_0}(z) \neq S^{n_0}(x_0)\}$$

$$L_z := \{\ S^i(y) \mid \exists p \in \mathbb{N},\ \exists u \in E_{z\alpha} \cup \{z\}.\ S^p(u) \neq S^i(y) \in H\ \}$$

$$k := max\{\ i + v(y) \mid S^i(y) \in L_z\ \}$$

Either of the two following cases holds:

- z does not occur at the left-hand side of any positive guard of α. If $E_{z\alpha} \neq \emptyset$ then there is a guard $S^p(u) = S^q(z) \in \alpha$ (recall that $S^{m_0}(z) \neq S^{n_0}(x_0)$ is a negative literal). Applying Lemma 2.33, on the path $\alpha \equiv \beta.(S^p(u) = S^q(z)).\gamma$, with the defined k above and the supposed valuation v, there is a number $l \in \mathbb{N}$ and a valuation v', such that:

$$\begin{aligned} &(i)\ v' \text{ satisfies } \alpha \\ &(ii)\ v'(u) = v(u) + l && \text{for each } u \in E_{z\alpha} \cup \{z\} \\ &(iii)\ v'(y) = v(y) && \text{for each } y \notin E_{z\alpha} \cup \{z\} \end{aligned}$$

Now define

$$l' := \begin{cases} l & \text{if } E_{z\alpha} \neq \emptyset \\ k+1 & \text{otherwise} \end{cases}$$

and

$$v''(y) := \begin{cases} v(y) + l' & \text{if } y \in E_{z\alpha} \cup \{z\} \\ v(y) & \text{otherwise} \end{cases}$$

Below we will show that v'' satisfies $(S^{m_0}(z) \neq S^{n_0}(x_0)).\alpha$:

- If $E_{z\alpha} = \emptyset$, note that z occurs in negative guards only. Also

  $$v''(y) \equiv \begin{cases} v(y) + k + 1 & \text{if } y \equiv z \\ v(y) & \text{otherwise} \end{cases}$$

  v'' satisfies each literal g which does not include z, since $v''(g) = v(g)$. Now we will show that it also satisfies every literal like $S^p(z) \neq S^q(y)$, which occurs on $\bar{\alpha} \cup \{S^{m_0}(z) \neq S^{n_0}(x_0)\}$:

  $$\begin{aligned} v'(S^p(z)) &= p + v'(z) \\ &= p + v(z) + k + 1 \\ &> k \\ &\geq v(S^q(y)) && \text{since } S^q(y) \in L_z \\ &= v'(S^q(y)) && v'(y) = v(y) \end{aligned}$$

- If $E_{z\alpha} \neq \emptyset$, then

  $$v''(y) \equiv \begin{cases} v(y) + l & \text{if } y \in E_{z\alpha} \cup \{z\} \\ v(y) & \text{otherwise} \end{cases}$$

  Therefore $v''(g) = v'(g)$, for each literal g in α, which means v'' satisfies α. Now for $S^{m_0}(z) \neq S^{n_0}(x_0)$: $x_0 \notin E_{z\alpha} \cup \{z\}$, because $x_0 \not\equiv z$, and also, by Lemma 2.28(ii), $x_0 \notin E_{z\alpha}$. Hence

  $$\begin{aligned} v''(S^{m_0}(z)) &= v(S^{m_0}(z)) + l \\ &> k && (l > k) \\ &\geq v(S^{n_0}(x_0)) && S^{n_0}(x_0) \in L_z \\ &= v''(S^{n_0}(x_0)) && x_0 \notin E_{z\alpha} \cup \{z\} \end{aligned}$$

- $S^m(z) = S^n(x)$ occurs positively on α, for some $x \in \bar{V}$ and some natural numbers m and n.
  - If $x \equiv 0$: then $S^m(z) = S^n(x) \equiv z = S^n(0)$ since $S^m(z) = S^n(x)$ is a simplified guard (Corollary 2.9). $S^{m_0}(z) = S^{n_0}(x_0) \prec S^m(z) = S^n(x)$, therefore $S^{m_0}(z) = S^{n_0}(x_0) \equiv z = S^{n_0}(0)$ according to Definition 2.13. v satisfies $z = S^n(0)$, so it also satisfies $z \neq S^{n_0}(0)$, by Lemma 2.29, so we are finished.
  - If $x \not\equiv 0$:
    * If $x_0 \equiv x$ then, regarding Lemma 2.29, v satisfies $S^{m_0}(z) \neq S^{n_0}(x_0)$, because it satisfies $S^m(z) = S^n(x)$.
    * If $x_0 \not\equiv x$: $\alpha \equiv \beta.(S^m(z) = S^n(x).\delta$ for some ordered paths β and δ. Using Lemma 2.33, for α and the given number k above, and the valuation v, there is a valuation v' and a natural number $l > k$, such that v' satisfies α, $v'(u) = v(u) + l$ if $u \in E_{x\alpha} \cup \{x\}$, and $v'(y) = v(y)$ if $y \notin E_{x\alpha} \cup \{x\}$. We will now show that v' is suitable.
      $x_0 \notin E_{x\alpha} \cup \{x\}$, because $x_0 \not\equiv x$, and for all $u \in E_{x\alpha}$ we have $u \succ x$ (by the definition of $E_{x\alpha}$) and $x \succ x_0$ (because $S^{m_0}(z) \neq S^{n_0}(x_0) \prec S^m(z) = S^n(x)$ by Definition 2.13). Therefore, if x_0 occurs on α, then $v'(x_0) = v(x_0)$. Otherwise we can define $v'(x_0) := v(x_0)$ without loss of generality, because v' still satisfies α. We will show that v' satisfies $S^{m_0}(z) \neq S^{n_0}(x_0)$ too:

$$
\begin{aligned}
v'(S^{m_0}(z)) &= v'(z) + m_0 \\
&= v(z) + l + m_0 && z \in E_{x\alpha} \\
&= v(S^{m_0}(z)) + l \\
&> k && l > k \\
&\geq v(S^{n_0}(x_0)) && S^{n_0}(x_0) \in L_z \\
&= v'(S^{n_0}(x_0))
\end{aligned}
$$

■

Corollary 2.35 *An immediate consequence of Theorem 2.34 is*

- $\top$ *is the only tautological* $(0, S, =)$*-E-OBDD.*
- $\bot$ *is the only contradictory* $(0, S, =)$*-E-OBDD.*
- *Every other* $(0, S, =)$*-E-OBDD is satisfiable.*

Proof. Each path in a tautological OBDD should end in a $\top$. Because if T is a tautological OBDD, containing a path α which ends in a $\bot$, then according to

Theorem 2.34, there is a valuation v which satisfies α. But then $v(T) = 0$, which is impossible since T is a tautology. Therefore, if T has more than one leaf, rule 3 of Definition 2.14 is applicable on a tautological OBDD which is not $\top$, and this contradicts the orderedness. So $T \equiv \top$. Similarly for a contradictory one. ■

In propositional methods, each formula has a unique OBDD representation. Our method does not provide such a property (see Example 2.36). One of the reasons the uniqueness is important is to check satisfiability and tautology of the formula. However this is also the main result of this chapter (see Corollary 2.35.)

Example 2.36 *Let $x \prec y \prec z \prec t$. The formula $\varphi : \; z = x \wedge z \neq y \wedge t \neq y$ has two normal forms (see Figure 2.6). The OBDD: $ITE(z = x, \top, ITE(z = y, ITE(t = y, \bot, \top), \top))$ represents φ and it is a normal form. On the other hand the BDD: $ITE(z = y, ITE(t = y, ITE(z = x, \top, \bot), \top), \top)$ also represents this formula. Below we make an OBDD out of this BDD:*

$$
\begin{aligned}
& ITE(z = y, ITE(t = y, ITE(z = x, \top, \bot), \top), \top) \\
\overset{8,\ z:=y}{\rightarrow}\ & ITE(z = y, ITE(t = y, ITE(y = x, \top, \bot), \top), \top) \\
\overset{6,1-5}{\rightarrow}\ & ITE(y = x, \top, ITE(z = y, ITE(t = y, \bot, \top), \top))
\end{aligned}
$$

This is an OBDD which is equivalent to φ as well, but not equal to the first OBDD.

2.5 Conclusion

We developed the theoretical basis for a decision procedure for boolean combinations of equations with zero and successor. First, a formula is transformed into a $(0, S, =)$-BDD. A term rewrite system on $(0, S, =)$-BDDs has been presented, which yields OBDDs (by definition). The system is proved to be terminating, and the normal forms have the desirable property that all paths are satisfiable. As a consequence, if a formula φ is a contradiction (i.e. equivalent to $\bot$), then it reduces to $\bot$. Similarly, tautologies reduce to $\top$. Therefore, our method can be used to decide tautology and satisfiability. Since the resulting OBDD is logically equivalent to the original formula, our method can also be used to simplify a formula. Although the resulting OBDDs are not unique, our method can also be used to check equivalence of formulas. In order to check whether φ and ψ are equivalent, we can check whether $\varphi \leftrightarrow \psi$ is a tautology.

Towards an Implementation. The basic procedure is presented as a term rewrite system. This is still a non-deterministic procedure, because a term can have more than one redex. By proving termination, we established that every strategy will yield an OBDD. However, some strategies might be more effective than others. In [ZvdP01] rewrite strategies are studied to compute OBDDs for plain propositional logic. In

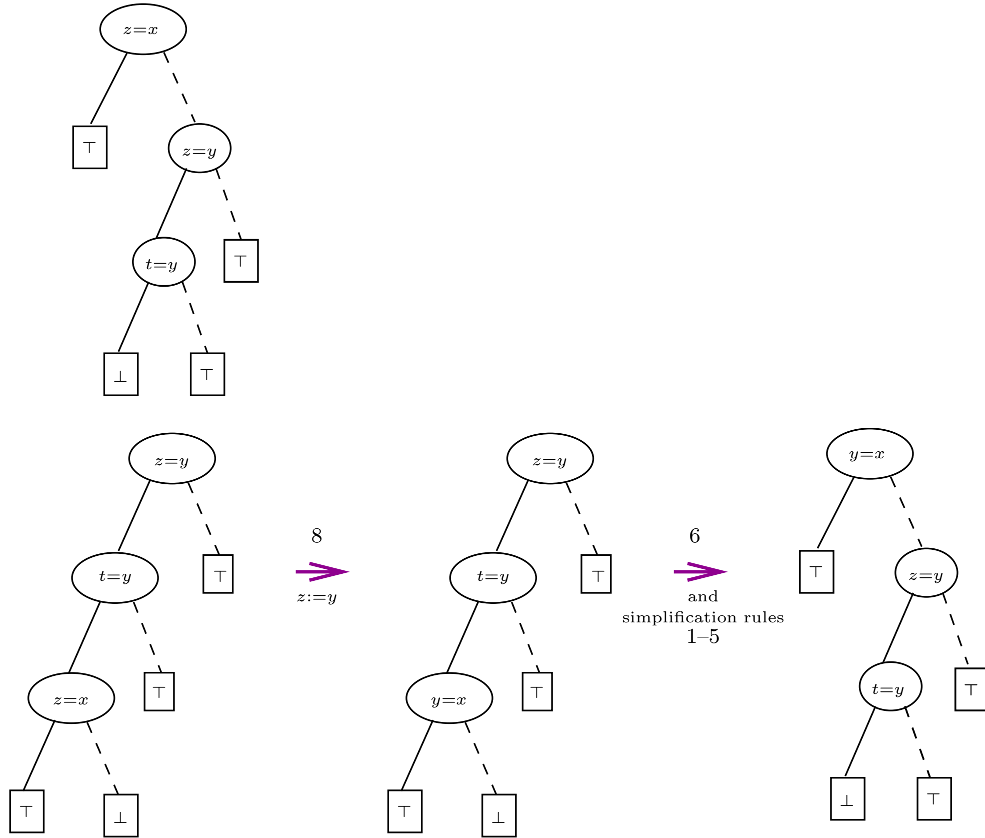

Figure 2.6: More than one OBDD representations Example 2.36

particular, it is shown how the usual efficient OBDD algorithms can be mimicked by a rewrite strategy. In [GvdP00] a concrete algorithm for EQ-BDDs was presented and proved correct. We have extended this algorithm for a different ordering, which will be introduced in the next chapter. The algorithm will be presented there as well. We have not yet studied particular strategies in the presence of zero and successor, nor implemented the procedure.

Our method (of this chapter) is also applied to the case of OBDDs for Equality logic with Uninterpreted Functions, by van de Pol and Tveretina [vdPT05, Tve05].

3

Representant-Ordered $(0, S, =)$-BDDs

In this chapter we provide an algorithm to verify formulas of the fragment of first-order logic, consisting of quantifier-free logic with zero, successor and equality. We first develop a rewrite system to extract an equivalent *Representant-Ordered* $(0, S, =)$-BDD from any given $(0, S, =)$-BDD. Then we show completeness of the rewrite system. Finally, based on the rewrite system, we make an algorithm that gives an equivalent Representant-Ordered $(0, S, =)$-BDD for any given $(0, S, =)$-BDD.

3.1 Introduction

In Chapter 2 we described a term rewrite system to transform $(0, S, =)$-BDDs into $(0, S, =)$-E-OBDDs. The approach in the current chapter is an alternative for making OBDDs.

There are three main technical differences between Chapter 2 and 3. Following an identical approach as the one introduced in [GvdP00], for the theory of equality logic with zero and successor, we came up with the method explained in this chapter. We place the smaller term of an equality in a $(0, S, =)$-BDD at the left-hand side. This minor change requires a different ordering on equalities and a different substitution rule in the term rewrite system, to provide termination. Second, in this chapter we provide an algorithm, by which we automatically obtain the equivalent so-called $(0, S, =)$-R-OBDD (Representant-Ordered $(0, S, =)$-BDD) to any given formula. Thirdly, the ideas of proofs are easier than those of the previous chapter.

The structure of this chapter is as follows: In Section 3.2 our transformation is presented, leading to the set of $(0, S, =)$-R-OBDDs. First a total and well-founded order on variables is assumed, and extended to a total well-founded order on equalities.

Then the rewrite system is presented. Finally, we prove termination and satisfiability over all paths. Section 3.28 presents an algorithm with the same result as the given term rewrite system. Section 3.4 describes some failed attempts. These are included in order to provide some insight in the subtleties of the method. Finally, Section 3.5 concludes with some remarks on implementation and possible applications.

In this chapter we reuse the preliminaries of Section 2.3. As we mentioned in Chapter 2, to start with defining an order on BDDs, first we assume an order on the set of variables. We consider a fixed total and well-founded order on V, and we extend it to terms by Definition 2.5.

3.2 Representant-Ordered $(0, S, =)$-BDDs

The first step to make a BDD ordered, is to simplify all its guards; in isolation. Here, simplification on guards will be done by Definition 3.1

In Section 3.2.1 we represent the new simplification system on guards. Next, in Definition 3.6 we define an ordering on set of guards, this ordering is different from Definition 2.13. Then we will have the basis to define the new term rewrite system in Definition 3.8. By applying this term rewrite system on any given $(0, S, =)$-BDD we obtain a *Representant Ordered* $(0, S, =)$-BDD, which we call $(0, S, =)$-R-OBDD.

Sections 3.2.2 shows that our term rewrite system is terminating, meaning that by applying it on any $(0, S, =)$-BDD we obtain a $(0, S, =)$-R-OBDD. Section 3.2.3 shows that all paths on a $(0, S, =)$-R-OBDD are satisfiable.

3.2.1 Definition of $(0, S, =)$-R-OBDDs

Recall that we have a fixed well-founded total order on V, and we extended this order to the set of terms W by Definition 2.5. We start by giving a new rewrite system to simplify guards.

Definition 3.1 *Suppose g is a guard. By $g\downarrow$ we mean the normal form of g obtained after applying the following rewrite rules on it:*

$$\begin{array}{rcll} x = x & \to & \top & \\ S(x) = S(y) & \to & x = y & \\ 0 = S(x) & \to & \bot & \\ x = S^{m+1}(x) & \to & \bot & \text{for all } m \in \mathbb{N} \\ t = r & \to & r = t & \text{for all } r, t \in W \text{ such that } r \prec t. \end{array}$$

We call g simplified if it cannot be further simplified, i.e. $g \equiv g\downarrow$. A $(0, S, =)$-BDD T is called simplified if all guards in it are simplified.

Next remark is an immediate result of this definition. It is similar to Definition 2.6, hence the proof is not repeated here.

Remark 3.2 *Suppose $g \in G$ is a guard which becomes g' after applying a certain rule of Definition 3.1 on it. Then g and g' are equivalent, i.e. they will have the same value under each valuation function.*

Among the above-mentioned rules, the last one will cause a major distinction with Definition 2.6. Here the smaller term is kept on the left-hand side of the guard. The next lemma shows the simplified guards; the difference with those of Chapter 2 is quite visible.

Lemma 3.3 *If g is a simplified guard, then it has one of the following shapes:*

- $S^m(0) = x$ *for some* $x \in V$
- $S^m(x) = S^n(y)$ *for some* $x, y \in V,\ x \prec y,\ m = 0$ *or* $n = 0$
- $\top$ *or* $\bot$

Note. As an immediate result of the previous lemma, each guard has only one normal form.

Below we define two notations (abbreviations) which will be used during the process of making a BDD ordered by our new term rewrite system.

Definition 3.4 *Suppose g is a simplified non-trivial guard, $y \in V$ and $t, r \in W$. We define:*

$$g|_{r=S^m(y)} := \begin{cases} (g{\uparrow^m} [S^m(y) := r]) \downarrow & \text{if } y \text{ occurs in } g \\ g & \text{otherwise} \end{cases}$$

$$g|_{t \neq r} := \begin{cases} \bot & \text{if } g \equiv (t = r) \downarrow \\ g & \text{otherwise} \end{cases}$$

The following remark demonstrates the soundness of the operations above:

Remark 3.5 *For any guard g and a positive natural number m, $g{\uparrow^m}$ and g are equivalent terms. Moreover suppose f is another guard. If f holds under a valuation v then g and $g|_f$ will be equivalent under v.*

Proof. The first part is trivial. Now for the second part of the remark, if f is positive then the proof is no different to that of the previous chapter (Remark 2.12). If f is $t \neq r$, then $v(t \neq r) = 1$ and hence $v(t = r) = 0$. Let us consider the following case distinction on g:

- $g \equiv (t = r) \downarrow$. Then

$$v((t = r) \downarrow) = v(t = r) \text{ (by Remark 3.2) } = 0 = v(\bot) = v(g|_{t \neq r}).$$

- $g \not\equiv (t = r)\downarrow$. Then it is trivial by its definition.

■

To obtain ordered BDDs, we still need to identify an ordering on the set of all simplified guards. The following order on simplified guards is the one we will build our new term rewrite system on.

Definition 3.6 (order on simplified guards) *We define a total order $\prec$ on simplified guards as below:*

- $\bot \prec \top \prec g$, *for all simplified guards g different from $\top$, $\bot$.*
- $(S^p(x) = S^q(y)) \prec (S^m(u) = S^n(v))$ *iff:*

$$\begin{aligned} i)&\ x \prec u \text{ or} \\ ii)&\ x \equiv u,\ p < m \text{ or} \\ iii)&\ x \equiv u,\ p \equiv m\ , y \prec v \text{ or} \\ iv)&\ x \equiv u,\ p \equiv m,\ y \equiv v,\ q < n \end{aligned}$$

According to this definition $(r_1 = t_1) \prec (r_2 = t_2)$ iff $(r_1, t_1) \prec_{lex} (r_2, t_2)$, in which $\prec_{lex}$ is a lexicographic order on quadruples of the total, well-founded orders $(\bar{V}, \prec) \times (\mathbb{N}, <) \times (\bar{V}, \prec) \times (\mathbb{N}, <)$, and therefore it is well-founded and total. This way without getting into the structures of the involved terms, only by knowing the order between them, one could determine the order of the guards.

Example 3.7 *Comparing the order above with that of previous chapter (Definition 2.6), one could see that these two sort guards differently. For instance let $g \equiv x = z$ and $f \equiv S^5(x) = y$ and $x \prec y \prec z$. With the order above we have: $g\downarrow$ is g itself and $f\downarrow$ is also f itself. moreover $g\downarrow \prec f\downarrow$. But with that of last chapter, we have: $g\downarrow \equiv z = x$ and $f\downarrow \equiv y = S^5(x)$, and $f\downarrow \prec g\downarrow$.*

Now we have all tools to introduce the term rewrite system, by which we will be able to reach a $(0, S, =)$-R-OBDD out of any $(0, S, =)$-BDD.

Definition 3.8 ($(0, S, =)$-R-OBDD) *An $(0, S, =)$-R-OBDD (Representant-Ordered $(0, S, =)$-BDD) is a simplified $(0, S, =)$-BDD (i.e. all its guards are simplified) which is a normal form with respect to the following term rewrite system:*

1. $ITE(\top, T_1, T_2) \rightarrow T_1$
2. $ITE(\bot, T_1, T_2) \rightarrow T_2$
3. $ITE(g, T, T) \rightarrow T$

4. $ITE(g, ITE(g, T_1, T_2), T_3) \to ITE(g, T_1, T_3)$

5. $ITE(g, T_1, ITE(g, T_2, T_3)) \to ITE(g, T_1, T_3)$

6. $ITE(g_1, ITE(g_2, T_1, T_2), T_3) \to ITE(g_2, ITE(g_1, T_1, T_3), ITE(g_1, T_2, T_3))$
   if $g_1 \succ g_2$

7. $ITE(g_1, T_1, ITE(g_2, T_2, T_3)) \to ITE(g_2, ITE(g_1, T_1, T_2), ITE(g_1, T_1, T_3))$
   if $g_1 \succ g_2$

8. *for every simplified* $(0, S, =)$*-BDD* C*:*
   $ITE(S^n(x) = S^m(y),\ C[g], T\ ) \to ITE(S^n(x) = S^m(y),\ C[\ g|_{S^n(x)=S^m(y)}], T\ )$
   if y *occurs in* g *and* $S^n(x) = S^m(y) \prec g$

One may notice that in rule 8, since $S^n(x) = S^m(y)$ is a simplified guard (by assumption), always one of m or n must be 0 (Lemma 3.3).

It is again obvious that the result of applying any rule on a simplified BDD is a simplified BDD. As an oppose to the Elimination-Ordered $(0, S, =)$-BDD, this time the reason we call the outcome Representant-Ordered $(0, S, =)$-BDD is that according to rule 8 the variable occurring in the left-hand side of a guard, in the $\top$-side sub BDD, represents the value of the term sitting in the right-hand side. It also has the first role to determine where the atom $S^n(x) = S^m(y)$ will sit, in the ordering process. Similar to the previous chapter, the input BDD will be equivalent to the result of applying any above rule on it:

Remark 3.9 *Suppose* $T \in B$ *is a* $(0, S, =)$*-BDD which becomes* T' *after applying any arbitrary rule of Definition 3.8 on it. Then* T *and* T' *are equivalent, i.e. they will have the same value under each valuation function. As a result each* $(0, S, =)$*-BDD is equivalent with its normal form (out of Definition 3.8).*

The next example demonstrates how the new term rewrite system works:

Example 3.10 *Let* $x \prec y \prec z$ *see Figure 3.1.*

$$
\begin{array}{cl}
 & ITE(S(y) = z, ITE(x = S^2(y), \top, \bot), \bot) \\
\xrightarrow{6} & ITE(x = S^2(y), ITE(S(y) = z, \top, \bot), ITE(S(y) = z, \bot, \bot)) \\
\xrightarrow{3} & ITE(x = S^2(y), ITE(S(y) = z, \top, \bot), \bot) \\
\xrightarrow{8} & ITE(x = S^2(y), ITE(\{S^3(y) = S^2(z)[S^2(y) := x]\}\downarrow, \top, \bot), \bot) \\
\overset{substitution}{\equiv} & ITE(x = S^2(y), ITE(\{S(x) = S^2(z)\}\downarrow, \top, \bot), \bot) \\
\equiv & ITE(x = S^2(y), ITE(x = S(z), \top, \bot), \bot)
\end{array}
$$

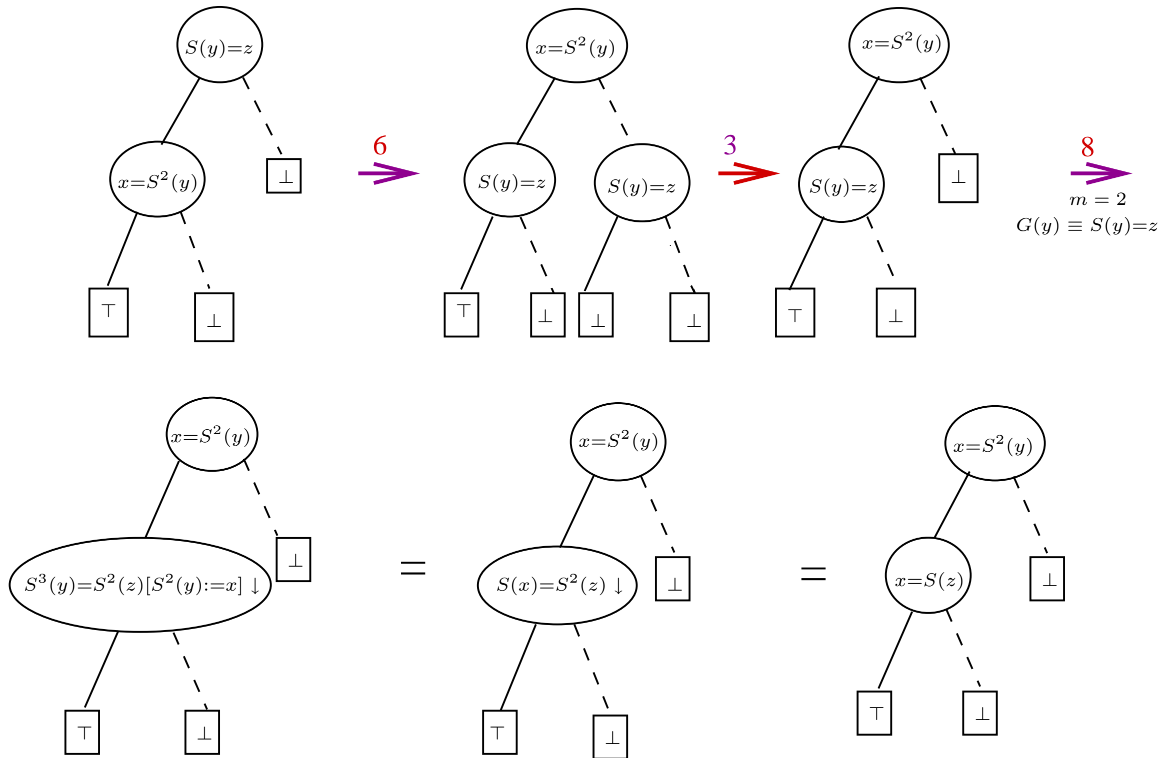

Figure 3.1: Derivation in Example 3.10

3.2.2 Termination

The first property to be checked for any given term rewrite system, is its termination; because only then we know that the rewriting process will always stop at some point.

To prove that our system is terminating we need some auxiliary lemmas first. We present them in Lemmas 3.11 and 3.12 below:

Lemma 3.11 *Let* $f \equiv S^n(x) = S^m(y)$ *and* $g \equiv S^k(v) = S^l(w)$*. If* $f \prec g$ *and* $f \equiv f\downarrow$ *and* $g \equiv g\downarrow$ *and* $y \in \{v, w\}$*, then* $g|_f \prec g$*.*

Proof.

- Case I: $y \equiv v$. Then $x \prec y\ (\equiv v) \prec w$, because f, g are simplified.
  $g|_f \equiv (g{\uparrow}^m\ [S^m(y) := S^n(x)])\downarrow \equiv (S^{k+n}(x) = S^{l+m}(w))\downarrow$, so

  $$g|_f \equiv x = S^{(l+m)-(k+n)}(w) \quad \prec g$$
  $$\text{or} \quad g|_f \equiv S^{(k+n)-(l+m)}(x) = w \quad \prec g$$
  $$\text{or} \quad g|_f \equiv \bot \quad \prec g$$

- Case II: $y \equiv w$.
  $g|_f \equiv (g{\uparrow}^m\ [S^m(y) := S^n(x)])\downarrow \equiv (S^{k+m}(v) = S^{l+n}(x))\downarrow$.

  If $x \equiv v$ then $g|_f \in \{\top, \bot\} \quad \prec g$

  else $x \prec v$ (because $f \prec g$), so

  $$g|_f \equiv x = S^{(k+m)-(l+n)}(v) \quad \prec g$$
  $$\text{or} \quad g|_f \equiv S^{(l+n)-(k+m)}(x) = v \quad \prec g$$
  $$\text{or} \quad g|_f \equiv \bot \quad \prec g$$

■

The idea of proof is similar to that of Lemma 2.20, however by repeating it we demonstrate that the different order will not affect this property to be true.

The next lemma expresses that after applying a rule on a BDD the outcome is always smaller, with respect to the $\succ_{rpo}$ order.

Lemma 3.12 *Each rewrite rule is contained in* $\succ_{rpo}$ (Definition 2.18).

Proof. Since rules 1–7 are similar to those in Definition 2.14, according to Lemma 2.22 they are contained in $\succ_{rpo}$.

Now since $S^n(x) = S^m(y)$ in rule 8 of Definition 3.8 is simplified, using Lemma 3.11, and the same technique as we used in Lemma 2.22, we obtain that rule 8 is also contained in this $\succ_{rpo}$ order. ■

Now we can prove that our term rewrite system (i.e. Definition 3.8) always terminates.

Theorem 3.13 *The rewrite system defined in Definition 3.8 is terminating, because $\succ_{rpo}$ is well-founded ([Zan03]).*

Proof. Similar to Theorem 2.23, using Lemma 3.12 we derive that the system is terminating. ■

Now that we know our system is terminating, we can immediately conclude that:

Corollary 3.14 *Every $(0, S, =)$-BDD is equivalent to at least one $(0, S, =)$-R-OBDD.*

Proof. Trivial. ■

3.2.3 Satisfiability of paths in $(0, S, =)$-R-OBDDs

From now on we will use the term OBDD to denote $(0, S, =)$-R-OBDD. We are going to prove that all paths in an OBDD are satisfiable. Here we use the same notations as Definition 2.25: α, β, γ represent paths and we write $\alpha.\beta$ for the concatenation of paths α and β.

The next two lemmas give syntactical properties on OBDDs, which can be used for proving satisfiability of each path in an OBDD.

Lemma 3.15 *Let $T \equiv ITE(S^m(x) = S^n(z), T_1, T_2)$ be a $(0, S, =)$-R-OBDD. Let α be a path in T_2 and $H = \{S^{j_i}(x) = r \mid 1 \leq i \leq k\}$ be the set of all positive guards on α which have x as their left-hand side variable. Then for each positive guard on α with a variable which occurs in an atom in H, we can conclude that the guard belongs to H.*

Proof. Let $E = \{x\} \cup \{Var(r) \mid S^{j_i}(x) = r \in H\}$ be the set of all variables occurring in H. Then we want to prove that for each positive guard on α with a variable in E, the guard belongs to H.

Write $H = \{\ S^{j_i}(x) = S^{l_i}(u_i) \mid 1 \leq i \leq k\ \}$. First note that $x \prec u_i$ for all $1 \leq i \leq k$, because H can only contain simplified guards.

- Guards $S^j(x) = S^l(u)$ are in H already, and guards $S^l(u) = S^j(x)$ cannot occur in α, because $u \prec x$ (guards are simplified), so such guards are smaller than $S^m(x) = S^n(z)$, contradicting the fact that T an OBDD.
- If there exists a guard like $t = r$ in α with $u_i \in \mathtt{Var}(t) \cup \mathtt{Var}(r)$ for some $1 \leq i \leq k$, then this guard can not occur below $S^{j_i}(x) = S^{l_i}(u_i)$ (rule 8 of Definition 2.14), so it should be above this guard
  - If $\mathtt{Var}(t) = u_i$ then, because it is placed above $S^{j_i}(x) = S^{l_i}(u_i)$ and T is ordered, $u_i \preceq x$, which is in contradiction with $x \prec u_i$.

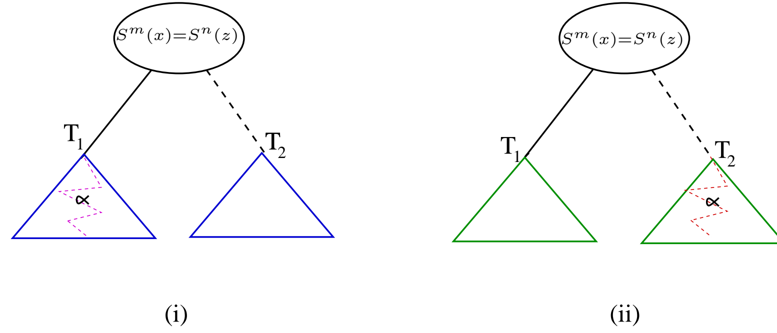

Figure 3.2: Satisfiable path α in OBDD T_1 or T_2

– If $\texttt{Var}(r) = u_i$ then, because it is placed above $S^{j_i}(x) = S^{l_i}(u_i)$, rule 8 of Definition 2.14 is applicable, which contradicts the fact that T an OBDD.

■

The next lemma says that in an OBDD, the left-most variable of each guard will not occur at the right-hand side of any guard underneath it.

Lemma 3.16 *Let $T \equiv ITE(S^m(x) = r, T_1, T_2)$ be a $(0, S, =)$-R-OBDD. Then for all guards $s = t$ occurring in T_1 or T_2 we have $x \not\equiv \texttt{Var}(t)$ (i.e. $t \not\equiv S^k(x)$ for any k).*

Proof. Let us assume that $S^k(y) = S^n(x)$ occurs in T_1 or T_2 for some k and n. Since T is an OBDD, each guard is simplified and so $y \prec x$. Therefore $S^k(y) = S^n(x) \prec S^m(x) = r$, and then one of the rules 6 or 7 of Definition 3.8 would be applicable; but this contradicts the fact that T is an OBDD. ■

Now we prove the second main theorem, which is satisfiability of each path in an OBDD.

Theorem 3.17 *Each path in a $(0, S, =)$-R-OBDD is satisfiable.*

Proof. We use induction over OBDDs. Suppose $T \equiv ITE(S^m(x) = S^n(z), T_1, T_2)$ is an OBDD, and each path which belongs to T_1 or T_2 is satisfiable. We will show that each path in T is satisfiable as well.

Suppose α is a satisfiable path in T_1 or T_2, so there is a valuation v which satisfies α. Let D be the set of elements in $\bar{V}$ that occur in α. We modify this valuation, in such a way that it satisfies $S^m(x) = S^n(z)$ — or its negation, depending on whether α is in T_1 or T_2 — and still satisfies α.

We continue the proof by considering a case distinction on whether α is sitting on the left- or right-hand side of $S^m(x) = S^n(z)$.

1. First, suppose α belongs to T_1 (see Figure 3.2(i)). Then $z \notin D$, because T is ordered and also rule 8 of Definition 3.8 would be applicable. Also $x \prec z$, because all guards are simplified, so $z \not\equiv 0$. From Lemma 3.3 we obtain: either $n = 0$, or $m = 0$ and $x \not\equiv 0$.
   a) If $n = 0$, define:

   $$v'(u) = \begin{cases} v(x) + m & \text{if } u \equiv z \\ v(u) & \text{otherwise} \end{cases}$$

   It can easily be derived that $v'(S^m(x) = z) = 1$, and since $z \notin D$, also for all $g \in \alpha$ we have $v'(g) = v(g) = 1$.

   b) If $n > 0$ and $m = 0$ and $x \not\equiv 0$, then by Lemma 3.16, $0 \notin D$. Now define:

   $$v'(u) = \begin{cases} v(x) & \text{if } u \equiv z \\ v(u) + n & \text{otherwise} \end{cases}$$

   $v'(x) = v'(S^n(z))$ obviously, so this valuation satisfies $x = S^n(z)$. Let an arbitrary guard $g \equiv S^p(u) = S^q(w)$ be given, such that $g \in \alpha$ (or $\neg g \in \alpha$). Since $u, w \in D$, and $0, z \notin D$, we have:

   $$\begin{aligned} v'(S^p(u)) = v'(S^q(w)) &\Longleftrightarrow v'(u) + p = v'(w) + q \\ &\Longleftrightarrow v(u) + n + p = v(w) + n + q \\ &\Longleftrightarrow v(u) + p = v(w) + q \\ &\Longleftrightarrow v(S^p(u)) = v(S^q(w)) \end{aligned}$$

   So v' satisfies g (or $\neg g$, respectively).

   Hence, when α belongs to T_1, v' satisfies α.

2. Next, suppose α belongs to T_2 (see Figure 3.2(ii)). Note that by Lemma 3.16, $r = S^j(x)$ can not occur in α (for any r and j). So let

   $$H = \{S^{j_i}(x) = S^{l_i}(u_i) \in \alpha \mid 1 \leq i \leq k\ \}$$

   to be the set of all (positive) guards in α in which x occurs (here x can be 0). Notice that $x \prec u_i$ for all $1 \leq i \leq k$. Now let

   $$m' = max\{j \mid S^j(x) \neq t \text{ occurs in } \alpha \text{ for some t}\}$$

(we set $m' = 0$ if this set is empty). Next, define for $y \in V$:

$$v'(y) = \begin{cases} v(y) & \text{if } y \in \{x, u_1, ..., u_k\} \\ v(y) + v(x) + m + m' + 1 & \text{otherwise} \end{cases}$$

We show that v' satisfies both $S^m(x) \neq S^n(z)$ and the path α.

- We first prove that $v'(S^m(x) \neq S^n(z)) = 1$. Note that $v'(S^m(x)) = m + v(x)$, regardless of whether $x \equiv 0$ or not. Also note that $z \not\equiv 0$. We distinguish the following cases:
  - If $z \not\equiv u_i$ for all $1 \leq i \leq k$, then $v'(S^n(z)) = v(z) + v(x) + m + m' + 1$, which is clearly greater than $v'(S^m(x))$, and therefore $v'(S^m(x)) \neq v'(S^n(z))$.
  - If $z \equiv u_i$ for some $1 \leq i \leq k$, then $S^{j_i}(x) = S^{l_i}(z) \in \alpha$. Since T_2 is an OBDD, either $m < j_i$ or $(m = j_i \wedge n < l_i)$.
    * If $m < j_i$, then by Lemma 3.3, $l_i = 0$:

$$\begin{aligned} v'(S^m(x)) &= v(x) + m \\ &< v(x) + j_i \\ &= v(u_i) + l_i \\ &= v(u_i) \\ &\leq v(u_i) + n \\ &= v'(u_i) + n \\ &= v'(S^n(u_i)) = v'(S^n(z)) \end{aligned}$$

    * If $m = j_i \wedge n < l_i$:

$$\begin{aligned} v'(S^m(x)) &= v(x) + m \\ &= v(x) + j_i \\ &= v(u_i) + l_i \\ &> v(u_i) + n \\ &= v'(u_i) + n \\ &= v'(S^n(u_i)) = v'(S^n(z)) \end{aligned}$$

  So in both cases $v'(S^m(x) \neq S^n(z)) = 1$

- Now we prove that v' satisfies all guards on the path α. Note that for r, t such that $r = t \in \alpha$, or $r \neq t \in \alpha$, we have $\texttt{Var}(t) \not\equiv 0$ by Lemma 3.3. Furthermore, if $\texttt{Var}(r) \equiv 0$, then $x \equiv 0$, by the ordering rules.
  - If $r = t \in \alpha$ then v satisfies $r = t$, hence $v(r) = v(t)$.

  * If $\texttt{Var}(r)$ or $\texttt{Var}(t) \in \{x, u_1, ...u_k\}$, then according to Lemma 3.15, $r = t \in H$, therefore $v'(r) = v(r)$ and $v'(t) = v(t)$, so $v'(r = t) = 1$.
  * Otherwise, note that $\texttt{Var}(r) \succ x$ and $\texttt{Var}(t) \succ x$, so both are non-zero. Hence $v'(r) = v(r) + v(x) + m + m' + 1$, and $v'(t) = v(t) + v(x) + m + m' + 1$ therefore $v'(r) = v'(t)$ and thus $v'(r = t) = 1$.
- If $r \neq t \in \alpha$ then v satisfies $r \neq t$, hence $v(r) \neq v(t)$.
  * If neither $\texttt{Var}(r)$ nor $\texttt{Var}(t)$ belongs to $\{x, u_1, ...u_k\}$, then both are non-zero, and $v'(t) = v(t) + v(x) + m + m' + 1$ and $v'(r) = v(r) + v(x) + m + m' + 1$, so $v'(r) \neq v'(t)$ because $v(r) \neq v(t)$, and hence $v'(r \neq t) = 1$.
  * If $\texttt{Var}(r)$ and $\texttt{Var}(t)$ both belong to $\{x, u_1, ...u_k\}$, then $r = t$ is of the form $S^j(z) = S^l(y)$ for $z, y \in \{x, u_1, ...u_k\}$. Hence by definition of v, $v'(r) = v'(S^j(z)) = v(S^j(z)) = v(r)$ and $v'(t) = v'(S^l(y)) = v(S^l(y)) = v(t)$. Therefore recalling that $v(r) \neq v(t)$ we have $v'(r) = v(r) \neq v(t) = v'(t)$. In other words $v'(r \neq t) = 1$.
  * If exactly one of $\texttt{Var}(r)$ or $\texttt{Var}(t)$ belongs to $\{x, u_1, ...u_k\}$, then one of these two cases holds:
    · This variable is x and since $\texttt{Var}(t) \neq x$ (by Lemma 3.16) $\texttt{Var}(r) = x$, therefore $r \equiv S^l(x)$ for some $l \in \mathbb{N}$ and thus

$$\begin{aligned} v'(r) &= v'(S^l(x)) \\ &= v'(x) + l \\ &= v(x) + l \\ &\leq v(x) + m' \quad \text{by definition of } m' \\ &< v(t) + v(x) + m + m' + 1 \\ &= v'(t) \end{aligned}$$

    So $v'(r) \neq v'(t)$ and hence $v'(r \neq t) = 1$
    · This variable is u_i for some $1 \leq i \leq k$. u_i will not occur in any literal below the positive guard $S^{j_i}(x) = S^{l_i}(u_i)$ (rule 8 of Definition 3.8), so that $r \neq t$ should be placed somewhere above this guard. But now $r = t$ is placed between $S^m(x) = S^n(z)$ and $S^{j_i}(x) = S^{l_i}(u_i)$, so $\texttt{Var}(r) \equiv x$ (by rules 6 and 7 of Definition 3.8). This contradicts the fact that only one of r, t contains u_i or x.

Hence, when α belongs to T_2, v' also satisfies α.

■

Since in OBDDs all paths are satisfiable, by a similar reasoning as Corollary 2.35, as an immediate result of Theorem 3.17 we can conclude the following:

Corollary 3.18

- $\top$ *is the only tautological* $(0, S, =)$*-R-OBDD.*
- $\bot$ *is the only contradictory* $(0, S, =)$*-R-OBDD.*
- *Every other* $(0, S, =)$*-R-OBDD is satisfiable.*

Proof. Similar to Corollary 2.35. ∎

3.3 An Algorithm

In this section we present an algorithm to transform any formula in equality logic with zero and successor to an equivalent OBDD. One could consider an algorithm which applies the rules of our term rewrite system one by one, on the given formula, until it reaches an OBDD. Although this is possible but it is not efficient, since in the process a lot of unnecessary cases will be checked on the formula, until it can reach a normal form. We instead extend the algorithm in [GvdP00], which is based in Shannon's expansion with the smallest equation $x = y$:

$$\varphi \iff (x = y \wedge \varphi|_{x=y}) \vee (x \neq y \wedge \varphi|_{x \neq y}).$$

Example 3.29 depicts how our algorithm works. We draw the reader's attention to the fact that the set of BDDs is a subset of the set of formulas. So in this section we may use BDDs wherever our scope is the set of all formulas.

In the first step, in order to simplify formulas as much as possible, we extend the reducing method in Definition 3.4 to all formulas:

Definition 3.19 *We extend the function of Definition 3.4, on formulas, for any simplified literal (guard) l:*

$$\begin{aligned}
(\neg\varphi)|_l &:= \neg(\varphi|_l) \\
(\varphi \wedge \psi)|_l &:= (\varphi|_l) \wedge (\psi|_l) \\
ITE(\varphi_1, \varphi_2, \varphi_3)|_l &:= ITE((\varphi_1)|_l, (\varphi_2)|_l, (\varphi_3)|_l)
\end{aligned}$$

As a result the corresponding remark (i.e. Remark 3.5) also can be extended to all formulas:

Remark 3.20 *Suppose l is a literal. If l holds under a valuation v then φ and $\varphi|_l$ will be equivalent under v.*

Proof. By induction over the structure of φ and using Remark 3.5, this can be easily checked. ■

Next lemma shows that after using an operation $|_l$ over a BDD, the outcome will not be bigger than the original BDD with respect to the $\succ_{rpo}$ order (Definition 2.18).

Lemma 3.21 *Let T be a simplified BDD. Suppose l is a simplified guard possibly occurring on T. If l is not bigger than the guards occurring on T then $T \succeq_{rpo} T|_l$.*

Proof. According to Lemma 3.11 and Definition 3.4, the guards do not get bigger. Now, by using induction over the structure of T and Definitions 3.19, the proof is trivial. ■

Next remark depicts a property of the definition which might be not so visible. This property will later be used to make some proof strategies easer.

Remark 3.22 *Let l be a simplified guard of the form $r = S^m(y)$ in which $y \in V$. Then $y \notin T|_l$.*

Proof. It is trivial by Definitions 3.19 and 3.4. ■

In the next definition we generalize the simplification method over guards in Definition 3.1 to all formulas, because in practice we will need to make the guards occurring inside BDDs (and formulas), smaller if that is possible:

Definition 3.23 *We extend the simplification rules of Definition 3.1 to all formulas below:*

$$g \to g\downarrow \quad \text{(if } g \text{ is not simplified)}$$
$$\neg g \to \neg(g\downarrow) \quad \text{(if } g \text{ is not simplified)}$$
$$(\varphi \wedge \bot) \to \bot \qquad (\bot \wedge \varphi) \to \bot$$
$$(\varphi \wedge \top) \to \varphi \qquad (\top \wedge \varphi) \to \varphi$$
$$(\neg\top) \to \bot \qquad (\neg\bot) \to \top$$
$$ITE(\top,\ \varphi,\ \psi) \to \varphi \qquad ITE(\bot,\ \varphi,\ \psi) \to \psi$$
$$ITE(g,\ \psi,\ \psi) \to \psi$$

$\varphi\downarrow$ represents a most simplified version of φ.

During the process of reducing BDDs, termination is a major concern to be dealt with. The next lemma will be used to prove the termination of our algorithm, after it is introduced.

Lemma 3.24 *Let T be a simplified BDD. If g is the smallest guard occurring in T, then $T \succ_{rpo} (T|_l)\downarrow$ for $l \in \{g, \neg g\}$.*

Proof. According to the last case of Definition 3.19, in order to calculate $T|_l$ we could apply the operation $|_l$ to its sub-trees. Let us consider a case distinction over l:

- $l \equiv g$. g occurs in T. Hence there is a sub-tree of T of the form $ITE(g, T_1, T_2)$. Let T' be one of these. Therefore:

$$\begin{aligned}
(T'|_g)\downarrow &\equiv ITE(g|_g, T_1|_g, T_2|_g)\downarrow && \text{(Definition 3.19)}\\
&\equiv (T_1|_g)\downarrow && \text{(Definition 3.23)}\\
&\preceq_{rpo} T_1|_g && \text{(using Lemma 3.12 for Definition 3.23)}\\
&\preceq_{rpo} T_1 && \text{(Lemma 3.21)}\\
&\prec_{rpo} T' && \text{(Lemma 2.18(I))}
\end{aligned}$$

  Above, $A \preceq_{rpo} B$ means $B \succeq_{rpo} A$. Now according to Remark 2.19, our original tree is bigger. Meaning that $T \succ_{rpo} (T|_l)\downarrow$. In this last conclusion we also used Lemma 3.12 and Lemma 3.21 implicitly.

- $l \equiv \neg g$. Similar.

■

The following function `sort` is meant to take the smallest guard occurring in a formula and bring it to the topmost place, and sort and simplify the formula afterward.

Definition 3.25 *We define a function* `sort` *on simplified formulas, which sorts and simplifies the given formula regarding the smallest contained guard.*

- $\mathtt{sort}(\bot) \equiv \bot$
- $\mathtt{sort}(\top) \equiv \top$
- *Let g be the smallest guard occurring positively or negatively in φ. Then*

$$\mathtt{sort}(\varphi) \equiv \begin{cases} \mathtt{sort}(\varphi|_g\downarrow) & \textit{if } \mathtt{sort}(\varphi|_g\downarrow) \equiv \mathtt{sort}(\varphi|_{\neg g}\downarrow) \\ ITE(g,\ \mathtt{sort}(\varphi|_g\downarrow),\ \mathtt{sort}(\varphi|_{\neg g}\downarrow)) & \textit{otherwise} \end{cases}$$

We recall the fact that BDDs are subsets of formulas. Therefore the function `sort` can be repeated (i.e. it can be applied on BDDs). The following remarks can immediately be deduced from this definition.

Remark 3.26 *The set of variables of* `sort`(φ) *is a subset of the set of variables of* φ*, for any formula* φ*.*

Proof. Trivial. ■

Remark 3.27 `sort`(φ) *is a BDD for any formula* φ*.*

Proof. This is trivial by the definition. ■

One application of `sort` does not always yield an OBDD (see e.g. Example 3.29). This fact forces us to repeatedly apply `sort` on the outcome till it is not applicable anymore. This is what we do in the algorithm below.

Definition 3.28 *The following algorithm, produces a $(0, S, =)$-R-OBDD for the input formulas:*

```
OBDD(φ)
 begin
     ψ:= φ↓ ;
     φ:= ⊥ ;
     while φ ≠ ψ do
         φ:= ψ ;
         ψ:= sort(ψ) ;
     od
     return ψ;
 end
```

Below by an example we show that it might be necessary to repeat the `sort` function:

Example 3.29 *Let $\varphi \equiv ITE(x = S(z), ITE(y = z, \top, \bot), \bot)$; we show how the* `OBDD` *algorithm finds an equivalent OBDD for this φ.*

φ is simplified already, so that $\psi = \varphi\downarrow = \varphi$. Now $\psi \neq \bot$, hence we must enter the **while**-loop: we first need to calculate $\texttt{sort}(\varphi)$. $x = S(z)$ is the smallest guard. $\texttt{sort}(\psi|_{x=S(z)}) \equiv ITE(x = S(y), \top, \bot)$ and $\texttt{sort}(\psi|_{x \neq S(z)}) \equiv \bot$. Hence

$$\begin{aligned}\texttt{sort}(\psi) &= ITE(x = S(z),\ \texttt{sort}(\psi|_{x=S(z)}),\ \texttt{sort}(\psi|_{x\neq S(z)}))\\ &= ITE(x = S(z),\ ITE(x = S(y), \top, \bot), \bot) \qquad \text{(above)}\end{aligned}$$

Now $\psi \neq \texttt{sort}(\psi)$, hence we must repeat the **while**-loop: $x = S(y)$ is the smallest guard. $\texttt{sort}(\psi|_{x=S(y)}) \equiv ITE(x = S(z), \top, \bot)$ and $\texttt{sort}(\psi|_{x\neq S(y)}) \equiv \bot$. Hence

$$\begin{aligned}\texttt{sort}(\psi) &= ITE(x = S(y),\ \texttt{sort}(\psi|_{x=S(y)}),\ \texttt{sort}(\psi|_{x\neq S(y)}))\\ &= ITE(x = S(y),\ ITE(x = S(z), \top, \bot), \bot) \qquad \text{(above)}\end{aligned}$$

Again $\psi \neq \texttt{sort}(\psi)$, hence we must repeat the **while**-loop: $x = S(y)$ is the smallest guard. $\texttt{sort}(\psi|_{x=S(y)}) \equiv ITE(x = S(z), \top, \bot)$ and $\texttt{sort}(\psi|_{x\neq S(y)}) \equiv \bot$. Hence

$$\begin{aligned}\texttt{sort}(\psi) &= ITE(x = S(y),\ \texttt{sort}(\psi|_{x=S(y)}),\ \texttt{sort}(\psi|_{x\neq S(y)}))\\ &= ITE(x = S(y),\ ITE(x = S(z), \top, \bot), \bot) \qquad \text{(above)}\end{aligned}$$

This time $\psi = \texttt{sort}(\psi)$, hence we must leave the **while**-loop, and stop with $\psi = ITE(x = S(y),\ ITE(x = S(z), \top, \bot), \bot)$ as the outcome.

We also present an example to show that our algorithm may cause different outputs (though equivalent) on equivalent inputs.

Example 3.30 *Let $\varphi \equiv x = z \vee y \neq z \vee y \neq t$ be the input formula where $x \prec y \prec z \prec t$. We make two equivalent formulas out of this input, and show that they will have different (but equivalent) OBDDs, out of the* `OBDD` *algorithm.*
One could represent φ as $ITE(x = z, 1, ITE(y = z, ITE(y = t, 0, 1), 1))$ or as $y = t \to (y = z \to x = z)$. Let us call the first representation ϕ_1 and the second ϕ_2. ϕ_2 is obviously equivalent to $y = t \to (y = z \to x = y)$. Now by applying the `OBDD` algorithm on them, we see that $\texttt{OBDD}(\phi_1) = \phi_1$ and on the second one we have

$$\texttt{OBDD}(\phi_2) = ITE(x = y, 1, ITE(y = z, ITE(y = t, 0, 1), 1))).$$

These two are obviously different.

Now we want to prove that the algorithm will always stop with an OBDD as the outcome (i.e. Theorem 3.33). The following two lemmas reveal some useful properties on `sort`, which will be used for the termination proof.

Lemma 3.31 *Let T be any simplified BDD. Then:*

1. $\texttt{sort}(T)\downarrow \equiv \texttt{sort}(T)$.
2. $T \succeq_{rpo} \texttt{sort}(T)$.

Proof.

1. We prove it with induction on $\succ_{rpo}$. If $T \in \{\top, \bot\}$ then the lemma holds. Suppose the lemma holds for any T' with $T \succ_{rpo} T'$.
   - If $\texttt{sort}(T) \equiv \texttt{sort}(T|_g\downarrow)$, then by Lemma 3.24 and the induction hypothesis, $\texttt{sort}(T)\downarrow \equiv \texttt{sort}(T)$.
   - If $\texttt{sort}(T) \equiv ITE(g,\ \texttt{sort}(T|_g\downarrow),\ \texttt{sort}(T|_{\neg g}\downarrow))$, then it is obvious that none of the rules of Definition 3.23 can be applied, since T is simplified, and both $\texttt{sort}(T|_g\downarrow)$ and $\texttt{sort}(T|_{\neg g}\downarrow)$ already establish the lemma. Thus $\texttt{sort}(T)\downarrow \equiv \texttt{sort}(T)$.
2. For the two trivial cases it obviously holds. Now using induction, we suppose that it holds for any $T \succ_{rpo} T'$. We show that it also holds for T:
   If $\texttt{sort}(T) \equiv \texttt{sort}(T|_g\downarrow)$ then regarding Lemma 3.24 and the hypothesis, $T \succeq_{rpo} \texttt{sort}(T)$.

If $\texttt{sort}(T) \equiv ITE(g,\ \texttt{sort}(T|_g \downarrow),\ \texttt{sort}(T|_{\neg g} \downarrow))$, moreover $T \equiv ITE(f, T_1, T_2)$ in which $f \succeq g$. Using Definition 2.18(II,III) it is obvious that either one of those two holds, and hence $T \succ_{rpo} \texttt{sort}(T)$ or $T \equiv \texttt{sort}(T)$. In any case $T \succeq_{rpo} \texttt{sort}(T)$.

■

One can easily check that the OBDD algorithm (i.e. Definition 3.28) stops as soon as $T \equiv \texttt{sort}(T)$. We claim that such a T is ordered:

Theorem 3.32 *Suppose T is a simplified $(0, S, =)$-BDD. If $T \equiv \texttt{sort}(T)$ then T is representant-ordered.*

Proof. We prove it inductively.

- $T \in \{\top, \bot\}$, then it is ordered.

- $T \equiv ITE(f, T_1, T_2)$. We assume that the theorem holds for any $T \succ_{rpo} T'$. Let g be the smallest guard occurring positively or negatively in T; we first show that $\texttt{sort}(T) \not\equiv \texttt{sort}(T|_g \downarrow)$. Let us distinguish two cases:

  - If $\texttt{sort}(T) \equiv \texttt{sort}(T|_g \downarrow)$, then

    $\texttt{sort}(T) \equiv T \succ_{rpo} T|_g \downarrow$ by Lemma 3.24. And by Lemma 3.31 $T|_g \downarrow \succeq_{rpo} \texttt{sort}(T|_g \downarrow) \equiv \texttt{sort}(T)$ (assumption). Hence $\texttt{sort}(T) \succ_{rpo} \texttt{sort}(T)$, which is a contradiction.

  - If $\texttt{sort}(T) \equiv ITE(g,\ \texttt{sort}(T|_g \downarrow),\ \texttt{sort}(T|_{\neg g} \downarrow))$, then

    since $T \equiv \texttt{sort}(T)$ we get that $f \equiv g$ and $T_1 \equiv \texttt{sort}(T|_g \downarrow)$ and $T_2 \equiv \texttt{sort}(T|_{\neg g} \downarrow)$. Below we first in two separate items prove that $T_i|_{g_i} \downarrow \equiv T_i$ for $i \in \{1, 2\}$, $g_1 \equiv g$ and $g_2 \equiv \neg g$.

    * If g occurs in T_2 then it is the smallest guard, since it is the smallest guard in T. Hence $\texttt{sort}(T|_{\neg g} \downarrow) \equiv \texttt{sort}(T_2) \succ_{rpo} T_2|_{\neg g} \downarrow$ (by Lemma 3.24) $\equiv T|_{\neg g} \downarrow$. So that $\texttt{sort}(T|_{\neg g} \downarrow) \succ_{rpo} T|_{\neg g} \downarrow$, which contradicts Lemma 3.31. Therefore g does not occur in T_2. Hence by Definition 3.4 and its extension Definition 3.19, $T_2|_{\neg g} \downarrow \equiv T_2 \downarrow$.
    * If $g \equiv r = S^m(y)$, then by Remark 3.22 $y \notin T|_g$, and so $y \notin T|_g \downarrow$ (obviously). Hence $y \notin \texttt{sort}(T|_g \downarrow)$ by Remark 3.26. Therefore by Definition 3.4, $T_1|_g \downarrow \equiv T_1 \downarrow$.

Now let $i \in \{1, 2\}$. Then

$$\begin{aligned}
T_i &\equiv \texttt{sort}(T|_{g_i}\downarrow) \\
&\equiv \texttt{sort}(T_i|_{g_i}\downarrow) && \text{(using definition of } T\text{)} \\
&\equiv \texttt{sort}(T_i\downarrow) && \text{(above)} \\
&\equiv \texttt{sort}(\texttt{sort}(T|_{g_i}\downarrow)\downarrow) \\
&\equiv \texttt{sort}(\texttt{sort}(T|_{g_i}\downarrow)) && \text{(by Lemma 3.31(1))} \\
&\equiv \texttt{sort}(T_i)
\end{aligned}$$

Using the induction hypothesis, T_1 and T_2 are ordered. Hence rules 1-7 of Definition 3.8 are not applicable on T, which means T will not be ordered only if the rule 8 is applicable . This rule can not be applied either, since $T_1|_g \equiv T_1$. Therefore T is ordered.

■

Now we can prove the main theorem which is termination of the algorithm:

Theorem 3.33 (Termination of the algorithm) *The algorithm given in Definition 3.28 is terminating, and* $\texttt{OBDD}(\varphi)$ *is a* $(0, S, =)$*-R-OBDD equivalent to* φ*, for any given formula* φ*.*

Proof. Regarding Remark 3.27 $\texttt{sort}(\varphi)$ will be a BDD, for any given formula φ. The $\succeq_{rpo}$ ordering is well-founded, therefore using Lemma 3.31(2) we know that after finitely many steps we will get $\texttt{sort}(\psi) \equiv \psi$, for some BDD ψ. Now using Theorem 3.32, ψ is ordered, and it is $\texttt{OBDD}(\varphi)$, regarding the algorithm.

■

Below we give another example to show how the `OBDD` algorithm works.

Example 3.34 *Let us consider the formula of Example 3.10:*

$$\varphi \equiv ITE(S(y) = z, ITE(x = S^2(y), \top, \bot), \bot).$$

We show how the `OBDD` *algorithm finds an equivalent OBDD for this* φ*.* φ *is simplified already, so that:* $\varphi\downarrow \equiv \varphi$*. Now* $\texttt{sort}(\varphi)$ *is:*

$$ITE(x = S^2(y),\ \texttt{sort}(\varphi|_{x=S^2(y)}),\ \texttt{sort}(\varphi|_{x\neq S^2(y)})).$$

The reader will be able to derive that:
$\texttt{sort}(\varphi|_{x=S^2(y)}) \equiv ITE(x = S(z), \top, \bot)$ *and*
$\texttt{sort}(\varphi|_{x\neq S^2(y)}) \equiv \bot$*. Replacing these two in the algorithm, we obtain*

$$\texttt{sort}(\varphi) \equiv ITE(x = S^2(y), ITE(x = S(z), \top, \bot), \bot).$$

It can also be checked that `sort(sort(`φ`))` $\equiv$ `sort(`φ`)`*, therefore* `sort(`φ`)` *is an equivalent OBDD to* φ*.*

In the next section we present some of our failed attempts before we found the appropriate ordering on the set of guards. As it was mentioned in the introduction (Chapter 1), our method for verification of extensions of equality logic, is not the same as encoding method. It is indeed in other direction. Having a proper ordering which could give us an idea of how we can do the extensions further to larger logics, was a question which was present along with the question of whether such an order exists even to the extended of equality logic with zero and successor. In the process of finding an answer, we faced at least two failed attempts. presenting them can give a vision on how such a basic order on terms, would be a fundamental criteria to have a general order on BDDs.

3.4 Failed Attempts

As shown in Sections 2.4 and 3.2, our main method is to extend a given ordering on variables to terms, and then lexicographically to guards, in such a way that we can prove *termination* (Theorems 2.23 and 3.13), which guarantees existence of OBDDs as normal forms, and *satisfiability* of paths (Theorems 2.34 and 3.17), which guarantees that contradictions and tautologies have unique OBDDs. The lexicographic extension of the term ordering to the guard ordering, as well as rules 1–8, are familiar from [GvdP00]. The creative parts are finding a good ordering on the terms and guards, and the idea of lifting equations. In this section we mention another approach, and two failed attempts, the first of which has non-terminating rewrite sequences, the second one has multiple contradictory OBDDs.

The variables come with a total order, say $y \succ x$. If we know that $y = x$, then y is eliminated in the $\top$-branch, by substituting the representant x for it. The solution that we have described in Chapter 2 orders the guards by grouping together the variables to be eliminated. In the alternative solution in this chapter, the representant variables are grouped together. This solution is closer to [GvdP00]. More precisely, the order on guards becomes: $S^p(x) = S^q(y) \prec S^m(u) = S^n(v)$ if and only if $(y, q, x, p)(\prec, <, \prec, <)_{lex}(v, n, u, m)$. For this ordering, the same results are proved, as we showed in this chapter.

Two failed attempts. We started our investigations with the ordering of Example 3.35. It was based on the observation that terms of the form $y = S^n(x)$ are easier to handle than $S^n(y) = x$. In the former case, all y's can be replaced by $S^n(x)$, while in the second case, replacing occurrences of $S^n(y)$ doesn't remove all occurrences of y. Later we solved this by lifting the equation. So we wanted to make terms with S-symbols smaller than terms without S-symbols. Obviously, the resulting ordering on guards is not well-founded. We tried to give an upper-bound on the number of

S-symbols that occur in a derivation, but this cannot be done.

Example 3.35 *Consider the following total ordering on variables and their successors:*

$$\ldots \prec S^2(x) \prec S^2(y) \prec \ldots \prec S(x) \prec S(y) \prec \ldots \prec x \prec y \prec \ldots$$

and its lexicographic extension to guards: $S^p(x) = S^q(y) \prec S^m(u) = S^n(v)$ *iff* $(q, y, p, x)(>, \prec, >, \prec)_{lex}(v, n, u, m)$. *Then consider the rewrite system of Definition 2.14, over this new ordering. Now look at the formula below:*

$$(y = S^2(x) \wedge z = S(y)) \vee (y \neq S^2(x) \wedge (S^2(z) = y \vee (S^2(z) \neq y \wedge z = S(y))))$$

In Figure 3.3 we show the first steps in a non-terminating rewrite sequence starting from this formula. We conjecture that this BDD has no normal form at all.

So unfortunately, this ordering can not be used, because it leads to non-termination, and the existence of OBDDs cannot be guaranteed. The first repair that comes into mind is reversing this order, so that it becomes well-founded. This led to our second try, in which terms without successors are smaller than terms having S-symbols.

Example 3.36 *Consider an alternative ordering on variables and their successors as below:*

$$x \prec y \prec \ldots \prec S(x) \prec S(y) \prec \ldots \prec S^2(x) \prec S^2(y) \prec \ldots \prec S^3(x) \prec \ldots$$

This order is extended lexicographically on guards: $S^p(x) = S^q(y) \prec S^m(u) = S^n(v)$ *iff* $(q, y, p, x)(<, \prec, <, \prec)_{lex}(v, n, u, m)$. *Next, we take rules 1–8 of Definition 2.14 w.r.t. to this new ordering. Now look at this formula:*

$$\varphi := S(y) \neq x \wedge S(x) = z \wedge S^2(y) = z$$

φ *is equivalent to* $\perp$*, but it has an OBDD (w.r.t. the new order) as drawn in Figure 3.4. This shows that a contradictory OBDD different from* $\perp$ *exists. The picture shows a path to* $\top$*, which is* un*satisfiable, so for this ordering, Theorem 2.34 wouldn't hold.*

Apparently, the occurrences of x in $S(y) = x$ and $S(x) = z$ are closely related, and should be treated in the same way. So we decided to change the ordering, so that all terms with x are smaller than all terms with y, etc. This led to the successful definition in Section 2.4. The price for allowing also terms of the form $S^n(y) = x$ is that in the substitution, we have to lift all occurrences of y to $S^n(y)$. This slightly complicates the formulation of rewrite rule 8.

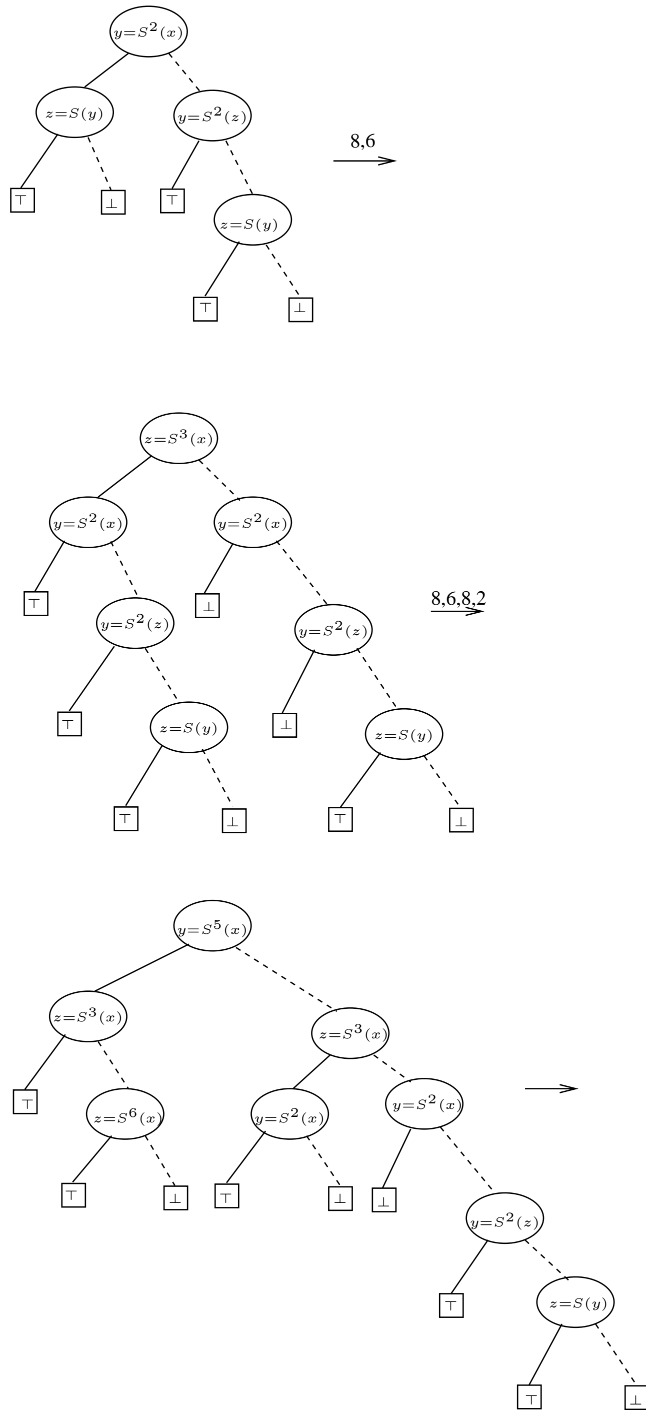

Figure 3.3: Derivation in Example 3.35

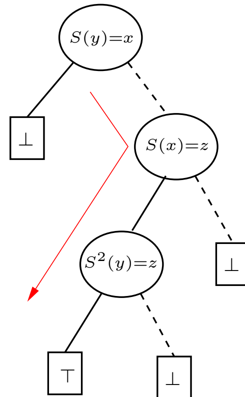

Figure 3.4: Unsatisfiable path for Example 3.36

3.5 Conclusion

We developed another decision procedure for boolean combinations of equations with zero and successor. Formulas are transformed into an equivalent $(0, S, =)$-BDD. Then the defined terminating rewrite systems on $(0, S, =)$-BDDs yields a $(0, S, =)$-R-OBDD out of them. The $(0, S, =)$-R-OBDD has the desirable property that all paths are satisfiable. Our method can be used to decide tautology and satisfiability. Then we provided a sound and complete algorithm, through which any formula is transformed to an equivalent $(0, S, =)$-R-OBDD. This algorithm gives an equivalent but not identical $(0, S, =)$-R-OBDD for a given $(0, S, =)$-BDD. This algorithm can also be presented for the term rewrite system of Chapter 2, however we just tried it for one as a demonstration. The idea should be the same.

Another line of research can be the extension of our result to other algebras. An interesting extension is the incorporation of uninterpreted functions directly. Other interesting extensions are the incorporation of addition (+), or an investigation of other free algebras (such as LISP-list structures based on null and cons). We did not find a solution in BDD style. The latter is what we will do in Chapters 4 and 5, using extensions of DPLL.

4

Ground Term Algebra

We make a decision procedure based on a generalization of the Davis-Putnam-Logemann-Loveland (DPLL) procedure, to decide satisfiability of formulas with equality, constructors (and recognizers in Chapter 5). We also make an algorithm through which a witness for any satisfiable formula can be obtained.

4.1 Introduction

In many algebraic systems, function symbols are divided in constructors and interpreted operations. The value of their presumed domains coincide with the ground terms built from constructor symbols. This is also the case with data specifications in μCRL [BvdPTZ04b]. Our scope in this chapter is formulas with constructor symbols only.

The *DPLL* procedure is a powerful and complete algorithm for checking satisfiability of propositional conjunctive normal form (CNF) formulas. DPLL is mainly based on the following splitting rule [DP60, DLL62]:
Given a formula φ, let l be a literal which occurs in φ, then φ is unsatisfiable if and only if $\varphi \cup \{\{l\}\}$ and $\varphi \cup \{\{\neg l\}\}$ are both unsatisfiable.

In this chapter we present a generalization of the DPLL procedure (GDPLL) for the logic of equality over infinite ground term algebras. This algorithm solves the satisfiability problem for a decidable fragment of quantifier-free first-order logic. It is an algorithm based on choosing an atom, adding it (or its negation) as a fact, reducing the intermediate formula and a satisfiability (stop) criterion. We show sufficient conditions on these basic steps under which GDPLL is sound and complete.

The formulas we work with are CNFs; we present them as sets of sets. The set

below is an example of the kind of formula that we deal with in this chapter:

$$\varphi = \{\{f(x) \approx y, g(y) \approx h(z,x)\}, \{x \not\approx z\}\}.$$

The interior sets are called *clauses*. So each formula is a set of clauses. Each clause is a set of *literals*. Literals are of the form $t \approx s$ or $t \not\approx s$, for some terms t and s.

4.2 Preliminaries

In this section we present some basic notations that are essential throughout this chapter.

4.2.1 Syntax

A signature Σ is a pair $(\texttt{Fun}, \texttt{Pr})$ of a set $\texttt{Fun} = \{f, g, h, \dots\}$ of *function symbols*, and a set $\texttt{Pr} = \{p, q, r \dots\}$ of *predicate symbols*. With each $f \in \texttt{Fun}$ and each $p \in \texttt{Pr}$ is associated a non-negative integer n, called the *arity* of f, resp. the arity of p.
Functions of arity zero are called *constant symbols*, we display them by $c, c_i, \dots$. Predicates of arity zero are called *propositional variables*. $\texttt{Var} = \{x, y, z, \dots\}$ represents the set of *variables*. $\texttt{Var}$, $\texttt{Fun}$ and $\texttt{Pr}$ are pairwise disjoint sets.

The set $\texttt{Term}(\Sigma, \texttt{Var})$ of *terms* over the signature Σ is inductively defined as follows.

- $\texttt{Var} \subseteq \texttt{Term}(\Sigma, \texttt{Var})$.
- For any $f \in \texttt{Fun}$ and all terms $t_1, \dots, t_n$ where n is the arity of f, we have $f(t_1, \dots, t_n) \in \texttt{Term}(\Sigma, \texttt{Var})$.

The set of *ground terms* $\texttt{Term}(\Sigma)$ is defined as $\texttt{Term}(\Sigma, \emptyset)$.

An *atom* a is defined to be an expression of the form $p(t_1, \dots, t_n)$, where the t_i are terms, and p is a predicate symbol of arity n. The set of atoms over the signature Σ and the set of variables $\texttt{Var}$ is denoted by $\texttt{At}(\Sigma, \texttt{Var})$, or simply by $\texttt{At}$ if Σ and $\texttt{Var}$ are clear.

A *literal* l is either an atom a or a negated atom $\neg a$. We define:

$$\neg l = \begin{cases} \neg a & \text{if } l = a \\ a & \text{if } l = \neg a \end{cases}$$

A literal l is *positive* if it is an atom, and it is *negative* if it is not positive (i.e. a negative atom). The set of all literals over the signature Σ is denoted by $\texttt{Lit}(\Sigma, \texttt{Var})$ or for simplicity by $\texttt{Lit}$. We denote by $\texttt{PLit}$ and $\texttt{NLit}$ respectively the set of all positive literals and the set of all negative literals. It is obvious that $\texttt{Lit} = \texttt{PLit} \cup \texttt{NLit}$.

A *clause* C is defined to be a finite set of literals. $\texttt{Cls}$ denotes the set of all clauses. The empty clause represents $\bot$.

A *conjunctive normal form* (CNF) is defined to be a finite set of clauses. Cnf refers to the set of all CNFs.

The following notations will be used frequently in this chapter.

Definition 4.1 *Assume* $\varphi \in \mathsf{Cnf}$ *and* $l \in \texttt{Lit}$*:*

- $\texttt{Var}(\varphi)$ *is the set of all variables occurring in* φ*.* $\texttt{Pr}(\varphi)$ *is the set of all predicate symbols occurring in* φ*.* $\texttt{At}(\varphi)$ *is the set of all atoms occurring in* φ *(similarly for* $\texttt{Term}(\varphi)$*,* $\texttt{Lit}(\varphi)$ *and* $\texttt{Cls}(\varphi)$*).* $\texttt{PLit}(\varphi)$*,* $\texttt{NLit}(\varphi)$ *are the set of all positive literals and the set of all negative literals in* φ *respectively.*

- $\varphi \wedge l$ *is a shortcut for* $\varphi \cup \{\{l\}\}$*.*

- $\varphi \wedge C$ *is a shortcut for* $\varphi \cup \{C\}$*.*

4.2.2 Semantics

Here we present some general semantics of this chapter.

Definition 4.2 *A* structure $\mathcal{D}$ *over a signature* $\Sigma = (\texttt{Fun}, \texttt{Pr})$ *is defined to consist of*

- *a non-empty set* D *called the* domain *of* $\mathcal{D}$*,*

- *for every* $f \in \texttt{Fun}$ *of arity* n *a map* $f_{\mathcal{D}} : D^n \to D$*,*

- *for every* $p \in \texttt{Pr}$ *of arity* n *a map* $p_{\mathcal{D}} : D^n \to \{\texttt{true}, \texttt{false}\}$*.*

Similar to the previous chapters, in order to deal with satisfiability of formulas, we need to introduce valuation functions. Here having a separate operation, as *interpretation*, which operates only on terms, would bring us more flexibility later on. We first define such a function:

Definition 4.3 *Let* $\mathcal{D}$ *be a structure. An* interpretation *is a map* $\sigma : \texttt{Var} \to D$*. Any interpretation* $\sigma : \texttt{Var} \to D$ *can be extended to a map* $\bar{\sigma} : \texttt{Term}(\Sigma, \texttt{Var}) \to D$ *by recursion, as follows:*

- $\bar{\sigma}(x) = \sigma(x)$ *if* $x \in \texttt{Var}$*.*

- $\bar{\sigma}(c) = c$ *if* c *is a constant symbol (i.e. function symbol of arity* 0*).*

- $\bar{\sigma}(f(t_1, \ldots, t_n)) = f_{\mathcal{D}}(\bar{\sigma}(t_1), \ldots, \bar{\sigma}(t_n))$*.*

From now on we use the notation σ *also for its extension* $\bar{\sigma}$*. We will use* σ*,* σ'*,* σ''*,* α*,* α'*,* α'' *as interpretations.*

Next we generalize the definition above, to all CNFs. Here is a general definition which can be used for any structure.

Definition 4.4 (Valuation) *Given an interpretation σ : $\mathtt{Var} \to D$, the* valuation *induced by σ is the mapping $\nu_\sigma : \mathtt{Lit} \cup \mathtt{Cls} \cup \mathtt{Cnf} \to \{\mathtt{false}, \mathtt{true}\}$ defined by*

$$\begin{aligned} \nu_\sigma(p(t_1,\ldots,t_n)) &= p_{\mathcal{D}}(\sigma(t_1),\ldots,\sigma(t_n)). \\ \nu_\sigma(\neg a) &= \neg \nu_\sigma(a). \\ \nu_\sigma(\{l_1,\ldots,l_m\}) &= \nu_\sigma(l_1) \vee \cdots \vee \nu_\sigma(l_m). \\ \nu_\sigma(\{C_1,\ldots,C_n\}) &= \nu_\sigma(C_1) \wedge \cdots \wedge \nu_\sigma(C_n). \end{aligned}$$

Next definition is quite depending on the intended structure.

Definition 4.5 *Given a structure $\mathcal{D}$ and a CNF φ in $\mathcal{D}$, we say φ is* satisfiable *in $\mathcal{D}$, if there is a valuation $\nu_\sigma : \mathtt{Lit} \cup \mathtt{Cls} \cup \mathtt{Cnf} \to \{\mathtt{false}, \mathtt{true}\}$ induced by an interpretation $\sigma : \mathtt{Var} \to D$, such that $\nu_\sigma(\varphi)$ is true in the structure.*

Example 4.6 *Suppose $\Sigma = (\{0\}, \{\approx\})$ is a signature. Let $\mathcal{D}$ be a structure over Σ, in which $\approx$ is the typical identity function, and with domain $D = \emptyset$. Then the CNF $\varphi = \{\{x \not\approx 0\}\}$ is not satisfiable in $\mathcal{D}$.*

Below we introduce a syntactical notation which is used very often in this chapter.

Definition 4.7 *Let φ be a CNF and l a literal. We define*

$$\varphi|_l = \{C - \{\neg l\} \mid C \in \varphi,\ l \notin C\}.$$

$\varphi|_l$ removes all the clauses in φ which contain l and removes $\neg l$ from the rest.

Intuitively, if a literal l is `true` then all the clauses which are including l are true. So these clauses do not impact the satisfiability of φ, and therefore they may be removed from the formula. Furthermore, $\neg l$ is `false` so it does not impact the satisfiability of the clauses involving it, so it may be removed from these clauses. The result would be the formula that we call $\varphi|_l$.

Example 4.8 *Consider the following formula:*

$$\varphi \equiv \{\{l, a\}, \{\neg l, b\}\}.$$

Then we will have:

$$\varphi|_l \equiv \{\{b\}\}.$$

It is obvious that if l is `true` then φ is `true` if and only if $\varphi|_l$ is `true`.

Lemma 4.9 *Let a be an atom. Then φ is satisfiable iff either $\varphi \wedge a$ or $\varphi \wedge \neg a$ is satisfiable.*

Proof.

- Suppose either $\varphi \wedge a$ or $\varphi \wedge \neg a$ is satisfiable. Let us assume $\varphi \wedge a$ is satisfiable. Then $\nu(\varphi \wedge a) = \texttt{true}$ for some valuation ν. Therefore by Definition 4.1 $\nu(\varphi \cup \{\{a\}\}) = \texttt{true}$. Hence by Definition 4.4 $\nu(\varphi) \wedge \nu(\{a\}) = \texttt{true}$. This results in $\nu(\varphi) = \texttt{true}$, which means that φ is satisfiable. With a similar proof strategy it can be shown that this holds also for the case of $\varphi \wedge \neg a$ being satisfiable.

- If φ is satisfiable then there is a valuation ν such that $\nu(\varphi) = \texttt{true}$. If $\nu(a) = \texttt{true}$ then similar to the above $\nu(\varphi \wedge a) = \texttt{true}$, otherwise $\nu(\neg a) = \texttt{true}$ and hence $\nu(\varphi \wedge \neg a) = \texttt{true}$.

■

4.3 Ground Term Algebra

In this section we show how to solve the satisfiability problem for CNFs over ground term algebras. In the sequel, we work with an arbitrary but fixed signature of the form $\Sigma = (\texttt{Fun}, \{\approx\})$. We assume that there exists at least one constant symbol (i.e. some $f \in \texttt{Fun}$ has arity 0), to avoid that the set $\texttt{Term}(\Sigma)$ of ground terms is empty. Later, we will also make the assumption that $\texttt{Term}(\Sigma)$ is infinite (i.e. at least one function symbol of arity > 0 exists, or the number of constant symbols is infinite).

Definition 4.10 (Ground term algebra)
Suppose $\Sigma = (\texttt{Fun}, \{\approx\})$ is a signature. We define a structure over this signature called the ground term algebra (associated with Σ) as follows:

- *$D = \texttt{Term}(\Sigma)$ is the set of ground terms,*

- *$f_{\mathcal{D}}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$ for all $f \in \texttt{Fun}$ and all $t_1, \ldots, t_n \in \texttt{Term}(\Sigma)$, where n is the arity of f,*

- *$\approx_{\mathcal{D}}$ is syntactic equality on terms.*

$f_{\mathcal{D}}$ coincides with applying function symbol f. Later on we will drop the subscript $\mathcal{D}$ from $f_{\mathcal{D}}$ and $\approx_{\mathcal{D}}$ for simplicity.

Below, we define a notation which would later on bring us more flexibility in expressing some terms.

Definition 4.11 *Let $[\,]$ be a new constant symbol which does not occur in Σ. A context F is a term in $\texttt{Term}(\Sigma \cup \{[\,]\})$, and it can be expressed as an incomplete term*

or a term with holes. However a context can have zero, one or more holes [Ter03], but for our purpose we would only consider contexts with (at most) one hole. Furthermore, $F[t]$ *denotes the result of replacing the hole with term* t*.*

To work with ground term algebras, first it is necessary to know the basic properties which hold in them.

Remark 4.12 *Let* Σ *be a signature. Then the following properties hold in the ground term algebra associated with it:*

1. *for all* $f, g \in \texttt{Fun}$ *with* $f \neq g$*: for all* $t_1, \dots, t_n, s_1, \dots, s_m \in \texttt{Term}(\Sigma)$:
$$f_{\mathcal{D}}(t_1, \dots, t_n) \neq g_{\mathcal{D}}(s_1, \dots, s_m)$$
2. *for all* $f \in \texttt{Fun}$*: for all* $t_1, \dots, t_n, s_1, \dots, s_n \in \texttt{Term}(\Sigma)$:
$$(t_1, \dots, t_n) \neq (s_1, \dots, s_n) \Rightarrow f_{\mathcal{D}}(t_1, \dots, t_n) \neq f_{\mathcal{D}}(s_1, \dots, s_n)$$
3. *for all contexts* $F \neq [\,]$*: for all* $t \in \texttt{Term}(\Sigma) : t \neq F[t]$

Below there is an example which represents how a formula in a ground term algebra looks like:

Example 4.13 *The set below is an example of a formula in a ground term algebra w.r.t. some* Σ

$$\varphi = \{\{f(t) \approx s, g(s) \approx h(t, f(t))\}, \{t \not\approx s\}\}.$$

In the next section we introduce some basic definitions and properties of interpretations and most general unifiers.

4.4 Substitutions and Most General Unifiers

We introduce here the standard definitions of substitutions and unifiers, taken from [LMM87, BN98].

Definition 4.14 *A* substitution *is a function* $\sigma : \texttt{Var} \to \texttt{Term}(\Sigma, \texttt{Var})$ *such that* $\sigma(x) \neq x$ *for only finitely many* xs*. We define the* domain $\mathsf{Dom}(\sigma)$ *as:*

$$\mathsf{Dom}(\sigma) = \{x \in \texttt{Var} \mid \sigma(x) \neq x\}.$$

If $\mathsf{Dom}(\sigma) = \{x_1, \dots, x_n\}$*, then we alternatively write* σ *as*

$$\sigma = \{x_1 \mapsto \sigma(x_1), \dots, x_n \mapsto \sigma(x_n)\}.$$

The variable range of σ *is*

$$\texttt{Var}(\sigma) = \bigcup_{x \in \mathsf{Dom}(\sigma)} \texttt{Var}(\sigma(x)).$$

Furthermore, with $\mathsf{Eq}(\sigma)$ *we denote the corresponding set of equation. That is*

$$\mathsf{Eq}(\sigma) = \{x_1 \approx \sigma(x_1), \ldots, x_n \approx \sigma(x_n)\},$$

and with $\neg\mathsf{Eq}(\sigma)$ *we mean the set of the negated equalities of* $\mathsf{Eq}(\sigma)$*. Hence*

$$\neg\mathsf{Eq}(\sigma) = \{x_1 \not\approx \sigma(x_1), \ldots, x_n \not\approx \sigma(x_n)\}.$$

Substitutions are extended to terms/literals/clauses as follows.

Definition 4.15 *We define the application of substitution* $(.)^\sigma$ *as below:*

$$\begin{aligned} x^\sigma &= \sigma(x) \\ f(t_1, \ldots, t_n)^\sigma &= f(t_1^\sigma, \ldots, t_n^\sigma) \\ (t \approx u)^\sigma &= t^\sigma \approx u^\sigma \\ (t \not\approx u)^\sigma &= t^\sigma \not\approx u^\sigma \\ \{l_1, \ldots, l_n\}^\sigma &= \{l_1^\sigma, \ldots, l_n^\sigma\} \\ \{C_1, \ldots, C_n\}^\sigma &= \{C_1^\sigma, \ldots, C_n^\sigma\} \end{aligned}$$

So, φ^σ is obtained from φ by replacing each occurrence of a variable x by $\sigma(x)$.

Definition 4.16 *The* composition $\sigma\rho$ *of substitutions* σ *and* ρ *is defined such that* $\sigma\rho(x) = \sigma(\rho(x))$*. A substitution* σ *is* more general *than a substitution* σ' *(notation:* $\sigma \lesssim \sigma'$*) if there is a substitution* δ *such that* $\sigma' = \delta\sigma$*. Furthermore, a substitution* σ *is* idempotent *if* $\sigma\sigma = \sigma$*.*

Definition 4.17 *A* unifier *or* solution *of a set* $S = \{s_1 \approx t_1, \ldots, s_n \approx t_n\}$ *consisting of a finite number of atoms, is a substitution* σ *such that* $s_i^\sigma = t_i^\sigma$ *for* $i = 1, \ldots, n$*.*
A substitution σ *is a most general unifier of* S *or in short* $\mathsf{mgu}(S)$*, if*

- σ *is a unifier of* S *and*
- $\sigma \lesssim \sigma'$ *for each unifier* σ' *of* S*.*

According to Definitions 4.14 and 4.17 we have $\mathsf{mgu}(\{x \approx x\}) = \emptyset$.

Definition 4.18 *An atom* $t \approx u$ *is in* solved form *if it is of the form* $x \approx u$*, where* x *is a variable,* u *is a term, and* x *does not occur in* u*. Otherwise it is non-solved. A literal* $\neg a$ *is in* solved form *if* a *is in solved form.*

In the sequel, we will use the following well-known facts on substitutions and unifiers (cf. [LMM87, BN98]).

Lemma 4.19

1. *A substitution* σ *is idempotent if and only if* $\mathsf{Dom}(\sigma) \cap \mathtt{Var}(\sigma) = \emptyset$.

2. *If a set* S *of atoms has a unifier, then it has an idempotent mgu.*

3. *If* $\sigma = \mathsf{mgu}(S)$ *and* σ *is idempotent, then* $\mathsf{Eq}(\sigma)$ *is in solved form, and logically equivalent to* S.

Notation and Conventions.

- A unit clause is a clause with only one literal.
- Let the CNF $\varphi = \{C_1, \ldots, C_n\}$ be a set of positive unit clauses. Then by $\sigma = \mathsf{mgu}(\varphi)$ we mean $\sigma = \mathsf{mgu}(\bigcup_{1 \leq i \leq n} C_i)$.
- We often simply write $\mathsf{mgu}(s \approx t)$ instead of $\mathsf{mgu}(\{s \approx t\})$.
- We use the notation $\mathsf{mgu}(S) = \perp$ if S has no unifier.
- From now on by an mgu we always mean an idempotent mgu, which exists only if there is a unifier (Lemma 4.19).

As a consequence of the above lemma and the conventions, if an mgu $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ then $x_i \notin \mathtt{Var}(t_j)$ for all $1 \leq i, j \leq n$.

After introducing some basic definitions and properties of interpretations and most general unifiers, we will define the building blocks of our decision algorithm (i.e. `GDPLL`), and prove that the obtained procedure is sound and complete.

4.5 GDPLL

Essentially, the DPLL procedure consists of the following three rules: the unit clause rule, the splitting rule, and the pure literal rule. Both the unit clause rule and the pure literal rule reduce the formula according some criteria. Most of the techniques relevant in the setting of the DPLL procedure are also applicable to `GDPLL`. `GDPLL` has a splitting rule, which carries out a case analysis with respect to an atom a. The current set of clauses φ splits into two sets: the one where a is `true`, and another where a is `false`.

The algorithm will be applied on reduced CNFs, which are the outcomes of applying a so-called *reduction system* on formulas. Our reduction system (Definition 4.20) is a set of transformation rules by which we obtain a simplified formula out of any given CNF. These rules can be applied on CNFs in any order. Therefore the system may not always produce a unique result.

Definition 4.20 (Reduction System) *We consider the following reduction rules, which should be applied repeatedly until* φ *cannot be modified any further.*

1. *If* $t \approx t \in C \in \varphi$ *then* $\varphi \longrightarrow \varphi - \{C\}$.

2. *If* $\bot \in \varphi$ *and* $\varphi \neq \{\bot\}$ *then* $\varphi \longrightarrow \{\bot\}$.

3. *If* $\varphi = \varphi_1 \uplus \{C \uplus \{t \not\approx u\}\}$, *and* $t \approx u$ *is non-solved, then let* $\sigma = \mathsf{mgu}(t \approx u)$ *and*
   - *if* $\sigma = \bot$, *then* $\varphi \longrightarrow \varphi_1$
   - *otherwise* $\varphi \longrightarrow \varphi_1 \cup \{C \cup \neg\mathsf{Eq}(\sigma)\}$.

4. *If* $\varphi_1 = \{C \mid C \in \varphi$ *is a positive unit clause*$\} \neq \emptyset$, *then let* $\sigma = \mathsf{mgu}(\varphi_1)$ *and*
   - *if* $\sigma = \bot$, *then* $\varphi \longrightarrow \{\bot\}$
   - *otherwise let* $\varphi_2 = \varphi - \varphi_1$, *and let* $\varphi \longrightarrow {\varphi_2}^{\sigma}$.

5. *If* $\varphi = \{\{\neg a\}\} \uplus \varphi_1$ *and* $a \in \mathsf{At}(\varphi_1)$ *then* $\varphi \longrightarrow \{\{\neg a\}\} \uplus \varphi_1|_{\neg a}$.

We recall that $\neg\mathsf{Eq}(\sigma)$ *is the set of the negations of all literals of* $\mathsf{Eq}(\sigma)$. *In other words it obtains by applying* $\neg$ *on all the literals of* $\mathsf{Eq}(\sigma)$.

A formula φ is *reduced* if none of the rules above is applicable on it. Also $\mathsf{Reduce}(\varphi)$ denotes a reduced form of φ with respect to this system. We can apply the rules on a given CNF φ in any order and this may cause different reduced forms for φ. This will be demonstrated by Example 4.24.

As opposed to in Chapters 2 and 3, in this chapter $x \approx y$ and $y \approx x$ are treated identically (i.e. we will not distinguish them as we did in Chapters 2 and 3); so a rule for symmetry is not needed. Note that a CNF is satisfiable if and only if the valuation satisfies at least one atom from every clause; respecting this, we introduce rules 1 (reflexivity) and 2 which are clear simplifications. Rule 3 replaces a negative equation by its solved form. Note that solving positive equations would violate the CNF structure, so this is restricted to unit clauses. Rules 4 and 5 above implement unit resolution adapted to the equational case. Positive unit clauses lead to substitutions. All positive units are dealt with at once, in order to minimize the calls to mgu and to detect more inconsistencies.

We will show that the rules are terminating, so at least one reduced form exists. The result of the reduction system is not always unique as we will show by Example 4.24 below, so $\mathsf{Reduce}(\varphi)$ is not uniquely defined. But any reduced form will suffice, as we will prove. Now we give some examples of reduction, and show which shape a reduced CNF may have.

Example 4.21 *Let* $\varphi = \{\{f(f(y)) \not\approx f(x)\}, \{x \not\approx x\}\}$*. Applying rule 3 above, on* $f(f(y)) \not\approx f(x)$ *we will have* $\sigma(x) = f(y)$*, therefore:*

$$\varphi \longrightarrow \{\{x \not\approx f(y)\}, \{x \not\approx x\}\}.$$

Once more applying the same rule on $x \not\approx x$*, we obtain:*

$$\varphi \longrightarrow \{\{x \not\approx f(y)\}, \{\}\}.$$

Since $\{\} = \bot$*, by rule 2 we get:*

$$\varphi \longrightarrow \{\bot\}.$$

Example 4.22 *The formula* φ *below is a reduced form, since no rule of Definition 4.20 is applicable on it:*

$$\varphi = \{\{x \not\approx f(y), z \approx g(x)\}, \{y \not\approx x\}\}.$$

Lemma 4.23 *Suppose* φ *is a reduced form. Then the following requirements hold:*

1. φ *contains no literal of the form* $t \approx t$*.*
2. *If* $\bot \in \varphi$ *then* $\varphi \equiv \{\bot\}$*.*
3. *All its negative literals are in solved form.*
4. φ *contains no positive unit clause.*
5. *If* $\varphi = \{\{\neg a\}\} \uplus \varphi_1$ *then* $a \notin \mathsf{At}(\varphi_1)$*.*

Proof. If φ does not satisfy one of the properties above, the corresponding rule can be applied. ■

Next, we give an example where $\mathsf{Reduce}(\varphi)$ is not uniquely defined.

Example 4.24 *Consider* $\varphi = \{\{x \not\approx f(a,b)\}, \{x \approx f(y,z)\}, \{y \approx a, x \approx f(a,b)\}\}$*. We show that using two different strategies, two distinct reduced forms for* φ *can be obtained:*

1. *One reduction:*

$$\begin{aligned} \varphi &\longrightarrow \{\{x \not\approx f(a,b)\}, \{x \approx f(y,z)\}, \{y \approx a\}\} && \text{(using 5)} \\ &\longrightarrow \{\{f(a,z) \not\approx f(a,b)\}\} && \text{(applying 4)} \\ &\longrightarrow \{\{z \not\approx b\}\} && \text{(using 3)} \end{aligned}$$

The result is a reduced form because no other rule is applicable on it.

2. *Another reduction:*

$$\begin{aligned}\varphi &\longrightarrow \{\{f(y,z) \not\approx f(a,b)\}, \{y \approx a, f(y,z) \approx f(a,b)\}\} && \text{(applying 4)}\\ &\longrightarrow \{\{y \not\approx a, z \not\approx b\}, \{y \approx a, f(y,z) \approx f(a,b)\}\} && \text{(using 3)}\end{aligned}$$

Which is again a reduced form.

We will show in Theorem 4.37 that given a ground term algebra with infinitely many closed terms, and a reduced formula φ, then φ is satisfiable in this ground term algebra if it contains no purely positive clauses (i.e. each clause has at least one negative literal). Below we introduce a predicate in order to distinguish these formulas.

Definition 4.25 *Let C be a clause. C is called* purely positive *if $C \cap \mathtt{NLit} = \emptyset$. Now let $\varphi \in \mathsf{Reduce}(\mathsf{Cnf})$; we define*

$\mathsf{SatCriterion}(\varphi) =$ *for all $C \in \varphi$, C is not a purely positive clause.*

Example 4.26 *Consider the following formula:*

$$\varphi \equiv \{\{t \not\approx f(x)\}, \{x \not\approx y, z \approx t\}\}$$

Then $t \not\approx f(x)$ is a negative literal in the first clause and $x \not\approx y$ is a negative literal in the second clause. Therefore none of the two clauses are purely positive, and hence we have $\mathsf{SatCriterion}(\varphi) = \mathtt{true}$.

Below we introduce an extension of DPLL algorithm over any ground term algebra. We call this algorithm `GDPLL`.

Definition 4.27 (`GDPLL` algorithm) *We present an algorithm by which the satisfiability of any CNF can be decided:*

```
GDPLL(φ)
   begin
      φ := Reduce(φ);
      if (⊥ ∈ φ) then return UNSAT;
      if (SatCriterion(φ)) then return SAT;
      choose a ∈ PLit(φ);
      if GDPLL(φ ∧ a) = SAT then return SAT;
      if GDPLL(φ|¬a ∧ ¬a) = SAT then return SAT;
```

```
    return UNSAT;
end
```

4.6 Termination

We will now prove termination of the reduction system and termination of the GDPLL procedure.

Below we define a function norm on formulas. This function associates with any CNF a pair of natural numbers. We will use this function as a measure to prove that the reduction system in Definition 4.20 is terminating. Below $\|S\|$ represents the cardinality of a finite set S.

Definition 4.28 *We define the following measures on formulas:*

$$\mathsf{pos}(\varphi) = \textit{number of occurrences of positive literals in } \varphi$$
$$\mathsf{neg}(\varphi) = \textit{number of occurrences of negative non-solved literals in } \varphi$$

To each formula φ, we correspond a pair of numbers, which we name $\mathsf{norm}(\varphi)$*:*

$$\mathsf{norm}(\varphi) = (\mathsf{pos}(\varphi) + \|\varphi\|,\ \mathsf{neg}(\varphi))$$

in which $\|\varphi\|$ is the cardinality of φ.

Theorem 4.29

(I) *The reduction system is terminating.*

(II) $\mathsf{pos}(\varphi)$ *does not increase during the reduction process on φ.*

Proof.

We prove I and II simultaneously by showing that after applying any step of the reduction system on a supposed formula, $\mathsf{norm}(\varphi)$ decreases with respect to the lexicographic order on pairs, and $\mathsf{pos}(\varphi)$ does not increase. So let $\varphi \longrightarrow \varphi'$; we distinguish cases according to the applied rule.

1. $\mathsf{pos}(\varphi') < \mathsf{pos}(\varphi)$ and $\|\varphi'\| < \|\varphi\|$, obviously.

2. $\|\varphi'\| = \|\{\bot\}\| = 1 < \|\varphi\|$, and $\mathsf{pos}(\varphi') \leq \mathsf{pos}(\varphi)$.

3. – if $\sigma = \bot$ then $\|\varphi'\|{=}\|\varphi\| - 1$ and
   $\mathsf{pos}(\varphi') \leq \mathsf{pos}(\varphi)$.
   – otherwise, $\mathsf{pos}(\varphi') = \mathsf{pos}(\varphi)$ and $\|\varphi'\| = \|\varphi\|$ but
   $\mathsf{neg}(\varphi') < \mathsf{neg}(\varphi)$ as we only count non-solved inequalities.

4. Let $\varphi = \varphi_1 \uplus \varphi_2$, where φ_1 is the non-empty set of positive unit literals in φ, and let $\sigma = \mathsf{mgu}(\varphi_1)$.
   – If $\sigma = \bot$ then $\|\varphi'\| = \|\{\bot\}\| = 1 \leq \|\varphi_1\| \leq \|\varphi\|$ and $\mathsf{pos}(\varphi') = \mathsf{pos}(\bot) < 1 \leq \mathsf{pos}(\varphi)$.
   – Otherwise $\|{\varphi_2}^\sigma\| = \|\varphi_2\| < \|\varphi_1\| + \|\varphi_2\| \leq \|\varphi\|$ and
   $\mathsf{pos}({\varphi_2}^\sigma) = \mathsf{pos}(\varphi_2) \leq \mathsf{pos}(\varphi)$.
5. Let $\varphi = \{\{\neg a\}\} \uplus \varphi_1$, with $a \in \mathsf{At}(\varphi_1)$.
   – if $a \in \mathtt{PLit}(\varphi_1)$ then using Definition 4.7

$$\mathsf{pos}(\varphi') = \mathsf{pos}(\varphi_1|_{\neg a}) \leq \mathsf{pos}(\varphi) - 1 < \mathsf{pos}(\varphi).$$

   We also have $\|\varphi'\| \leq \|\varphi\|$.
   – otherwise $\neg a \in \mathtt{Lit}(\varphi_1)$ and hence

$$\begin{aligned} \|\varphi'\| &\leq \|(\varphi_1|_{\neg a})\| + 1 \\ &< \|\varphi_1\| + 1 && \text{(Definition 4.7)} \\ &= \|\varphi\| \end{aligned}$$

   We also have $\mathsf{pos}(\varphi') \leq \mathsf{pos}(\varphi)$.

■

Next, we identify an order on formulas. However it differs from the other orderings we introduced in previous chapters, but this is the order which depicts the desired property, termination of the algorithm.

Definition 4.30 *Given two reduced formulas φ and ψ, we define $\psi \prec \varphi$ if $\mathsf{pos}(\psi) < \mathsf{pos}(\varphi)$.*

Having the above definition, we are now able to prove that the formulas get smaller in the process of reducing them.

Theorem 4.31 (Termination criterion) $\mathsf{Reduce}(\varphi \wedge (t \approx u)) \prec \varphi$ *and* $\mathsf{Reduce}(\varphi|_{t \not\approx u} \wedge (t \not\approx u)) \prec \varphi$ *for any reduced formula φ and literal $t \approx u \in \mathtt{PLit}(\varphi)$.*

Proof. We prove each one separately:

- First we prove that $\mathsf{Reduce}(\varphi \wedge (t \approx u)) \prec \varphi$.
  - If $\mathsf{Reduce}(\varphi \wedge (t \approx u)) = \{\bot\}$, then $\mathsf{Reduce}(\varphi \wedge (t \approx u)) \prec \varphi$ obviously since $t \approx u \in \mathtt{PLit}(\varphi)$.
  - Suppose $\mathsf{Reduce}(\varphi \wedge (t \approx u)) \neq \{\bot\}$. Then since φ is a reduced formula, rule 5 of Definition 4.20 cannot be applied on φ; so since $t \approx u \in \mathtt{PLit}(\varphi)$, then $\{t \not\approx u\} \notin \varphi$. Hence rule 5 is not applicable on $\varphi \wedge (t \approx u)$ either. Moreover, rules 1, 2 and 3 cannot be applied because of the theorem assumptions. Therefore the first step to reduce $\varphi \wedge (t \approx u)$, according to Definition 4.20, will be rule 4, as follows:

    $$\varphi \wedge (t \approx u) = \varphi \uplus \{\{t \approx u\}\} \text{ (Lemma 4.23(4))}$$

    Let $\sigma = \mathsf{mgu}(t \approx u)$; if $\sigma = \bot$ then $\varphi \uplus \{\{t \approx u\}\} \longrightarrow \{\bot\}$ so $\mathsf{Reduce}(\varphi \wedge (t \approx u)) = \{\bot\} \prec \varphi$. Otherwise since $t \approx u \in \mathtt{PLit}(\varphi)$, then $t^\sigma \approx u^\sigma \in C \in \varphi^\sigma$, where $t^\sigma = u^\sigma$ because $\sigma = \mathsf{mgu}(t \approx u)$. Assume that $\varphi^\sigma = \varphi_0 \rightarrow \varphi_1 \rightarrow \ldots \rightarrow \varphi_{n+1} = \mathsf{Reduce}(\varphi^\sigma)$ is the reduction sequence by which we obtain $\mathsf{Reduce}(\varphi^\sigma)$ from φ^σ.
    Applying any rule of Definition 4.20 on φ_0, $t^\sigma \approx u^\sigma$ will be either removed or replaced by a similar one $t^\rho \approx u^\rho$. By Lemma 4.23(1), φ_{n+1} does not contain any literal of the shape $w \approx w$.
    Since φ_0 contains at least one literal of that shape ($t^\sigma \approx u^\sigma$), there exists a $0 \leq j \leq n+1$ such that φ_j has a literal of the form $w \approx w$, and φ_{j+1} does not have any. According to Theorem 4.29(II), the number of occurrences of the positive literals does not increase during the reduction process, therefore $\mathsf{pos}(\varphi_j) \leq \mathsf{pos}(\varphi_{j+1}) - 1$. Hence $\mathsf{pos}(\varphi_0) < \mathsf{pos}(\varphi_{n+1})$, again by Theorem 4.29(II).
- Since according to Definition 4.7 $\{t \not\approx u\} \notin \varphi|_{t\not\approx u}$, it follows that

  $$\varphi|_{t\not\approx u} \wedge (t \not\approx u) = \varphi|_{t\not\approx u} \uplus \{\{t \not\approx u\}\}.$$

Furthermore, according to the same definition $t \approx u \notin \mathtt{PLit}(\varphi|_{t\not\approx u})$. Hence

$$\begin{aligned} \mathsf{pos}(\mathsf{Reduce}(\varphi|_{t\not\approx u} \wedge (t \not\approx u))) &\leq \mathsf{pos}(\varphi|_{t\not\approx u} \uplus \{\{t \not\approx u\}\}) && \text{(Theorem 4.29(II))} \\ &\leq \mathsf{pos}(\varphi) - 1 && (t \approx u \notin \mathtt{PLit}(\varphi|_{t\not\approx u})) \end{aligned}$$

■

Corollary 4.32 $\mathtt{GDPLL}(\varphi)$ *is always terminating.*

Proof. This is an immediate result of *Termination criterion* (Theorem 4.31). ■

In the next section we show that the reduction system preserves satisfiability. Also we prove a criterion which introduces and proves a condition under which any reduced formula is satisfiable.

4.7 Correctness Properties

Lemma 4.33 *Suppose σ is an interpretation which satisfies the literal l. Then given a formula φ, σ satisfies φ if and only if σ satisfies $\varphi|_l$.*

Proof. We prove each side separately:

- If σ satisfies φ then by Definition 4.7 we must prove that σ satisfies $C - \{\neg l\}$ for any $C \in \varphi$, where $l \notin C$. σ does not satisfy $\neg l$, since it satisfies l. Moreover σ satisfies C, since it satisfies φ. Hence σ satisfies $C - \{\neg l\}$. Therefore σ satisfies $\varphi|_l$.
- If σ satisfies $\varphi|_l$, then by Definition 4.7, we only need to show that σ satisfies every clause C of φ containing l. σ satisfies l, therefore it will also satisfy any clause C containing l.

■

Next Lemma describes that each formula is equi-satisfiable with its reduction forms. This is a very important property, since we will later on search for (un)satisfiability of a reduced form of a given formula than the original one itself.

Theorem 4.34 (Reduce criterion) *Given a ground term algebra $\mathcal{D}$ and a formula φ in it, φ is satisfiable iff $\mathsf{Reduce}(\varphi)$ is satisfiable.*

Proof.
We check in any step of the reduction that φ is satisfiable iff the result is satisfiable. So assume that $\varphi \to \varphi'$; we now make a distinction on the five rules of Definition 4.20.

1. It is obvious that α satisfies φ if and only if α satisfies φ', for each interpretation α.
2. Both are unsatisfiable.
3. (a) Suppose α satisfies φ. Then in the first case obviously α satisfies φ', which is $\varphi - \{C\}$. In the second case also α satisfies φ' because $t \not\approx u$ is replaced with the negation of its unifier, which is equivalent by Lemma 4.19(3).

   (b) Let α satisfy φ'. Either $\varphi' = \varphi - \{C\}$ and $t \not\approx u \in C \in \varphi$ is a tautology, or φ' is obtained from φ by replacing $t \not\approx u$ with $\neg\mathsf{mgu}(t \approx u)$, see Lemma 4.19(3). In any case α satisfies φ.

4. Let φ_1 be the non-empty subset of positive unit clauses in φ, and let $\varphi = \varphi_1 \uplus \varphi_2$.
   (a) If $\varphi' = \{\bot\}$ then φ' is unsatisfiable, and according to this rule φ_1 has no unifier. Therefore φ is unsatisfiable too.
   (b) Suppose α satisfies φ. So α satisfies φ_1, that is α is a unifier for φ_1. Since σ is an idempotent mgu of φ_1 then by Definition 4.17 there is a substitution α' for φ_1 such that $\alpha = \alpha'\sigma$. As a result, since σ is idempotent $\alpha\sigma = \alpha'\sigma\sigma = \alpha'\sigma = \alpha$. Hence for each term t we have $\alpha(t^\sigma) = \alpha\sigma(t) = \alpha(t)$. Therefore α satisfies ${\varphi_2}^\sigma$, since it satisfies φ_2.
   On the other hand, suppose α satisfies ${\varphi_2}^\sigma$, define:
   $$\alpha'(y) = \begin{cases} \alpha(y) & \text{if } y \in \mathtt{Var}({\varphi_2}^\sigma) \\ \alpha(\sigma(y)) & \text{otherwise} \end{cases}$$
   α' satisfies φ.
5. Is obvious by Lemma 4.33.

■

Although according to the above theorem, each formula is equi-satisfiable with its reduced ones, yet these two may have different set of satisfying valuations:

Example 4.35 *Let $\varphi = \{\{x \approx y\}, \{x \not\approx c\}\}$. We show that there is a valuation which satisfies a reduced form of it, but not the formula itself.*
Formula $\psi = \{\{y \not\approx c\}\}$ is a reduced form of φ (by rule 4 of Definition 4.20). Let c_1 be a constant symbol which is not identical to c, then define α as:

$$\alpha(z) = \begin{cases} c_1 & \text{if } z = y \\ c & \text{otherwise} \end{cases}$$

then (the valuation induced by)α satisfies ψ but it does not satisfy φ obviously.

Before presenting a criterion for satisfiability of CNFs, we define a function called *depth*. This fuction is introduced to halp us in the proof of the next theorem:

Definition 4.36 *Given a term t we define $depth(t)$ to be the depth of a term t defined as follows:*

$$\begin{aligned} depth(x) &= 0 \\ depth(c) &= 0 && \text{(if } c \text{ is a constant symbol)} \\ depth(f(t_1,\ldots,t_n)) &= 1 + max_{1\le i\le n} depth(t_i) && \text{(if } n \ge 1) \end{aligned}$$

Theorem 4.37 (SAT criterion) *Suppose $\mathcal{D}$ is a ground term algebra with infinitely many closed terms. Then a reduced formula φ is satisfiable if* $\mathsf{SatCriterion}(\varphi)$ *is* `true`.

Proof. $\mathcal{D}$ has infinitely many closed terms, hence Σ contains at least one constant symbol c. Suppose φ is a CNF formula which has the properties of the theorem, i.e. φ is reduced and φ has no purely positive clause (in particular, $\bot \notin \varphi$). Let $n = \|\varphi\|$. Then each clause of this formula has a negative literal of the form $x_i \not\approx t_i$, for $1 \leq i \leq n$, which is also solved by Lemma 4.23. It suffices to provide an interpretation σ which satisfies all these negative literals, because then each clause is satisfiable with that σ, which implies that φ is satisfiable. We distinguish two cases:

- $\mathcal{D}$ has at least one function symbol g, of arity m, greater than zero.
  Suppose c is a constant symbol in $\mathcal{D}$. We define a context $F[\,]$ as:
  $F[\,] = g([\,],\underbrace{c,\ldots,c}_{m-1 \text{ times}})$. Now define a number $M = 1 + max_{1 \leq i \leq n} depth(t_i)$.
  Let $F^M[\,]$ be M-fold application of F on $[\,]$. Consider the following interpretation:

  $$\sigma(x) = \begin{cases} F^{M.i}(c) & \text{if } x = x_i, \text{ for some } 1 \leq i \leq n \\ c & \text{otherwise} \end{cases}$$

  We claim that σ satisfies $x_i \not\approx t_i$ for each $1 \leq i \leq n$:
  Indeed, note that $depth(\sigma(x_i)) = M.i$. Moreover, if $depth(t_i) = 0$, then $depth(\sigma(t_i)) = M.j$ with $0 \leq j \leq n$ and $i \neq j$ ($x_i \neq t_i$ because φ is reduced). Otherwise, $depth(\sigma(t_i)) = M.k + l$ for some $k \geq 0$, and $0 < l \leq depth(t_i) < M$. In both cases, $depth(\sigma(x_i)) \neq depth(\sigma(t_i))$.

- $\mathcal{D}$ has no non-constant function symbols. Therefore each of its negative literals are of the shape $x \not\approx t$, in which $x \neq t$ and t is a variable or a constant symbol, since $x \not\approx t$ is a solved atom. Define:

  $$V_\varphi = \{x \mid x \text{ is a variable occurring in } \varphi\}$$
  $$C_\varphi = \{c \mid c \text{ is a constant symbol occurring in } \varphi\}$$

  We know that the two given sets are of finite cardinality. Without loss of generality suppose that $V_\varphi = \{x_1, x_2, \ldots, x_n\}$, for some $n \in \mathbb{N}$. Since $\mathcal{D}$ has infinitely many constant symbols, there exists a set $\mathcal{C} = \{c_1, c_2, \ldots, c_{n+1}\}$, of $n+1$ distinct constant symbols of $\mathcal{D}$, such that $C_\varphi \cap \mathcal{C} = \emptyset$. Define:

  $$\sigma(x) = \begin{cases} c_i & \text{if } x = x_i, \text{ for some } x_i \in V_\varphi \\ c_{n+1} & \text{otherwise} \end{cases}$$

  Now $x_i \not\approx t$ has one of the following shapes:

– $x_i \not\approx x_j$. Then σ satisfies it since $\sigma(x_i) \neq \sigma(x_j)$.
– $x_i \not\approx c$. Then σ satisfies it since $\sigma(x_i) = c_i \neq c = \sigma(c)$, because $C_\varphi \cap \mathcal{C} = \emptyset$.

■

The following example shows the importance of having infinitely many closed terms in Theorem 4.37.

Example 4.38 *Suppose* $\Sigma = (\mathtt{Pr}, \mathtt{Fun})$ *is a signature where* $\mathtt{Fun} = \{0, 1\}$ *consists of two function symbols of arity zero. Then a formula* $\varphi = \{\{x \not\approx 0\}, \{x \not\approx 1\}\}$ *is unsatisfiable in any ground term algebra over* Σ*.*

Since there are only two constant symbols and no other function symbols, therefore the set of closed terms is consisting of only two elements, for any ground term algebra over this signature. Hence any term is either 0 or 1. So that φ is unsatisfiable.

It is obvious that if there were more than two closed terms in the ground term algebra then the formula above would be satisfiable.

4.8 Correctness of GDPLL

We can now combine the lemmas on the basic blocks in order to conclude correctness of `GDPLL` for ground term algebras.

Theorem 4.39 (Soundness and completeness)
Let $(\mathtt{Fun}; \approx)$ *be a signature with infinitely many ground terms. Let* $\mathcal{D}$ *be its ground term algebra. Let* φ *be a CNF. Then*

- φ *is satisfiable iff* $\mathtt{GDPLL}(\varphi)$ *returns* SAT*.*
- φ *is unsatisfiable iff* $\mathtt{GDPLL}(\varphi)$ *returns* UNSAT*.*

Proof.

- We prove the implications separately:
  – Suppose $\varphi \in \mathsf{Cnf}$ is satisfiable. Then by the *Reduce criterion* (Theorem 4.34) $\mathsf{Reduce}(\varphi)$ is satisfiable. Hence $\perp \notin \mathsf{Reduce}(\varphi)$. Assume (induction hypothesis) that the theorem holds for all ψ such that $\mathsf{Reduce}(\psi) \prec \mathsf{Reduce}(\varphi)$ (Definition 4.30). Now
    * If $\mathsf{SatCriterion}(\mathsf{Reduce}(\varphi))$ holds then $\mathtt{GDPLL}(\varphi)$ is SAT.

* If $\mathsf{SatCriterion}(\mathsf{Reduce}(\varphi))$ does not hold, then by Definition 4.25 there exists a clause $C \in \mathsf{Reduce}(\varphi)$ such that $C \cap \mathtt{NLit} = \emptyset$. Since $C \neq \bot$ ($\bot \notin \mathsf{Reduce}(\varphi)$) and all of its literals are positive. As a result $\mathtt{PLit}(\varphi) \neq \emptyset$. By *Termination criterion* (Theorem 4.31), for all $a \in \mathtt{PLit}(\varphi)$ we have $\mathsf{Reduce}(\mathsf{Reduce}(\varphi) \wedge a) \prec \mathsf{Reduce}(\varphi)$, and also $\mathsf{Reduce}(\mathsf{Reduce}(\varphi)|_{\neg a} \wedge \neg a) \prec \mathsf{Reduce}(\varphi)$. Now, using Lemma 4.9 and Lemma 4.33, either of $\mathsf{Reduce}(\varphi) \wedge a$ or $\mathsf{Reduce}(\varphi)|_{\neg a} \wedge \neg a$ is satisfiable. Therefore, applying the induction hypothesis, the corresponding $\mathtt{GDPLL}$ will return SAT. Hence the algorithm itself will return SAT. This means that $\mathtt{GDPLL}(\varphi)$ returns SAT.

– Suppose $\mathtt{GDPLL}(\varphi)$ is SAT. According to the *Reduction criterion* (Theorem 4.34), we only need to prove that $\mathsf{Reduce}(\varphi)$ is satisfiable.
Now by the $\mathtt{GDPLL}$ algorithm, (Definition 4.27), $\mathsf{SatCriterion}(\mathsf{Reduce}(\varphi))$ holds, or there exists an $a \in \mathtt{PLit}(\varphi)$ such that $\mathtt{GDPLL}(\mathsf{Reduce}(\varphi) \wedge a)$ is SAT, or $\mathtt{GDPLL}(\mathsf{Reduce}(\varphi)|_{\neg a} \wedge \neg a)$ is SAT. Using induction, we suppose that the theorem holds for all ψ such that $\mathsf{Reduce}(\psi) \prec \mathsf{Reduce}(\varphi)$ (Definition 4.30).

* if $\mathsf{SatCriterion}(\mathsf{Reduce}(\varphi))$ holds, then using *SAT criterion* (Theorem 4.37) we have that $\mathsf{Reduce}(\varphi)$ is satisfiable.

* if $\mathtt{GDPLL}(\mathsf{Reduce}(\varphi) \wedge a)$ is SAT, then since $\mathsf{Reduce}(\mathsf{Reduce}(\varphi) \wedge a) \prec \mathsf{Reduce}(\varphi)$ by Theorem 4.31, we can apply the induction hypothesis. So $\mathsf{Reduce}(\varphi) \wedge a$ is satisfiable. Hence $\mathsf{Reduce}(\varphi)$ is satisfiable.

* if $\mathtt{GDPLL}(\mathsf{Reduce}(\varphi)|_{\neg a} \wedge \neg a)$ is SAT. Then $\mathsf{Reduce}(\mathsf{Reduce}(\varphi)|_{\neg a} \wedge \neg a) \prec \mathsf{Reduce}(\varphi)$ by Theorem 4.31, hence by applying the induction hypothesis we get that $\mathsf{Reduce}(\varphi)|_{\neg a} \wedge \neg a$ is satisfiable. Therefore $\mathsf{Reduce}(\varphi)$ is satisfiable.

• This is a result of termination (Corollary 4.32) and the first item of this theorem.

■

In the next section we present two algorithms to get a reduced form of CNFs, and also to find a witness which shows their satisfiability when they are satisfiable.

4.9 The Witness

First we introduce an algorithm which gives us a reduced form out of any given CNF; this is what we call the Reduce algorithm below. This algorithm applies the reduction

system of Definition 4.20 on given CNFs. The outcome of this algorithm is a formula which is a reduced form, and an interpretation α. This interpretation will be used later on to produce a witness for the given formula.

```
Reduce(φ : Cnf):Cnf =
    begin
        α := ∅;
        while either of the rules of Definition 4.20 are applicable on φ then
            choose one of the applicable rules;
            if the rule is one of the rules 1, 3 or 5 then apply that on φ and
                call the result φ;
            if the rule is rule 2 then return (⊥, ∅);
            if the rule is rule 4 then
                let φ1 = {C | C ∈ φ is a positive unit clause};
                let φ2 be such that φ = φ1 ⊎ φ2;
                let σ = mgu(φ1);
                if σ = ⊥ then return(⊥, ∅);
                otherwise φ := φ2^σ and α := σα;
            end if
        end while
        return(φ, α);
    end
```

In Example 4.40 we use the CNF of Example 4.24 to show how the Reduce algorithm works.

Example 4.40 *Consider* $\varphi = \{\{x \not\approx f(a,b)\}, \{x \approx f(y,z)\}, \{y \approx a, x \approx f(a,b)\}\}$. *We apply the* Reduce *algorithm on this CNF:*

In the first step we have $\alpha := \emptyset$, then since rules 5 and 4 are applicable on φ we can get into the **while** loop. Let us choose rule 4, then according to the algorithm, we get: $\varphi_1 := \{\{x \approx f(y,z)\}\}$,
$\varphi := \{\{x \approx f(y,z)\}\} \uplus \{\{x \not\approx f(a,b)\}, \{y \approx a, x \approx f(a,b)\}\}$,
$\sigma := \{x \mapsto f(y,z)\}$,
$\varphi := \{\{f(y,z) \not\approx f(a,b)\}, \{y \approx a, f(y,z) \approx f(a,b)\}\}$ and
$\alpha := \sigma\alpha = \sigma\emptyset = \{x \mapsto f(y,z)\}$,

Now rule 3 is the only applicable rule on φ. So by applying this rule we get $\varphi := \{\{y \not\approx a, z \not\approx b\}, \{y \approx a, f(y,z) \approx f(a,b)\}\}$. We can see that none of the rules are applicable on φ anymore, therefore the algorithm stops and returns

$$(\{\{y \not\approx a, z \not\approx b\}, y \approx a, f(y,z) \approx f(a,b)\}\}, \{x \mapsto f(y,z)\}).$$

Obviously depending on the rules we choose to rewrite φ with, we may derive different results from this algorithm.

Now we introduce some primarily algorithms below, in order to use them for making a second `GDPLL` algorithm which not only will let us know whether a CNF is satisfiable or not, but also will give us a witness for those CNFs which are satisfiable.

Depending on whether the ground term algebra has any non-constant function symbol or not, we introduce the next two algorithms:

Definition 4.41 (Func and Cons algorithms) *Given any two sets X and Y of variables and a natural number d_X, if the ground term algebra has at least one non-constant function symbol, then the* **Func** *algorithm will return an interpretation for given CNFs.*

$$\mathbf{Func}(\varphi : \mathsf{Cnf}, X, Y, d_X) \colon \mathtt{Var} \to \mathtt{Term}(\Sigma, \mathtt{Var}) =$$

begin

$\alpha := \emptyset;$

choose $g \in \mathtt{Fun}$;

choose a constant symbol c;

let $F[\,] = g([\,], \underbrace{c, \ldots, c}_{\text{arity}(g)-1 \text{ times}});$

$M := d_X + 1;$

$\sigma := \{x \mapsto F^{jM}(c) \mid (x, j) \in X\} \cup \{z \mapsto c \mid z \notin Y\};$

$\alpha := \sigma\alpha;$

return α;

end

If the ground term algebra has only constant function symbols, then the **Cons** *algorithm will return an interpretation for CNFs.*

$$\mathbf{Cons}(\varphi : \mathsf{Cnf}, X, Y, d_X) \colon \mathtt{Var} \to \mathtt{Term}(\Sigma, \mathtt{Var}) =$$

begin

$\alpha := \emptyset;$

let $n = \|\varphi\|$;

let $\mathcal{C} = \{c_1, c_2, \ldots, c_{n+1}\}$ of $n + 1$

distinct constant symbols which do not occur in φ;

$\sigma := \{x \mapsto c_j \mid (x, j) \in X\} \cup \{z \mapsto c_{n+1} \mid z \notin Y\};$

$\alpha := \sigma\alpha;$

return α;

end

The next algorithm uses the **Func** and **Cons** algorithms to give an interpretation for CNFs with a certain property:

Definition 4.42 (SatCriterion algorithm) *The* SatCriterion *algorithm below, calculates two sets X and Y and also a number d_X for the given CNF, and then depending*

on the ground term algebra decides which of the two algorithms **Func** *and* **Cons** *have to be applied on the CNF. The outcome of this algorithm is an interpretation:*

SatCriterion(φ : Cnf): $\mathtt{Var} \to \mathtt{Term}(\Sigma, \mathtt{Var}) =$
begin
$Y := \emptyset$;
$X := \emptyset$;
$d_X := 0$;
$i := 0$;
while $\varphi \neq \emptyset$
choose $C \in \varphi$ and $x \not\approx u \in C$;
$\varphi := \varphi - \{C\}$;
$d_X := max(d_X, depth(u))$;
if $x \notin Y$ **then**
$i := i + 1$;
$Y := Y \cup \{x\}$;
$X := X \cup \{(x, i)\}$;
end if
end while
if there is a function symbol **then**
Func(φ, X, Y, d_X);
else Cons(φ, X, Y, d_X);
end

Now we have all the building blocks of the main algorithm. Below the `GDPLL` algorithm is introduced which gives us a witness for satisfiable CNFs, beside announcing the satisfaction of the CNF.

Definition 4.43 (The `GDPLL` algorithm) *Given any CNF φ the extended* `GDPLL` *algorithm will let us know whether the CNF is satisfiable, and if that is so, it will present a witness which makes the formula true:*

GDPLL(φ : Cnf): $(\{\mathsf{UNSAT}, \mathsf{SAT}\}, \mathtt{Var} \to \mathtt{Term}(\Sigma, \mathtt{Var})) =$
begin
$(\varphi, \alpha) :=$ Reduce(φ);
if $(\bot \in \varphi)$ **then** return (UNSAT, $\emptyset$);
else if φ has no purely positive clause **then**
return (SAT, SatCriterion(φ));
end if
else choose $a \in \mathtt{PLit}(\varphi)$;
if GDPLL$(\varphi \wedge a) = (\mathsf{SAT}, \alpha)$ **then** return (SAT, α);
else if GDPLL$(\varphi|_{\neg a} \wedge \neg a) = (\mathsf{SAT}, \alpha)$ **then** return (SAT, α);

else return $(\mathsf{UNSAT}, \emptyset)$;
end

As an example we use the reduced CNF of Example 4.13 to show how this algorithm works:

Example 4.44 *Consider* $\varphi = \{\{f(x) \approx y, g(y) \approx h(z,x)\}, \{x \not\approx z\}\}$. *We show that this CNF is satisfiable and moreover we give an interpretation which satisfies it.*

Let us apply the `GDPLL` algorithm on this CNF. In the first step we have $(\varphi, \alpha) := \mathsf{Reduce}(\varphi) = \varphi$, since none of the rules of the reduction system (Definition 4.20) are applicable.

Now we can see that $\bot \notin \varphi$ and also φ has a purely positive clause. Therefore according to the algorithm we need to choose a positive literal $a \in \mathtt{PLit}(\varphi)$. Let us choose $a := g(y) \approx h(z,x)$. Then we need to calculate $\mathtt{GDPLL}(\varphi \wedge (g(y) \approx h(z,x)))$ where $\varphi \wedge a = \{\{f(x) \approx y, g(y) \approx h(z,x)\}, \{x \not\approx z\}, \{g(y) \approx h(z,x)\}\}$. The first immediate step is to calculate $\mathtt{Reduce}(\varphi \wedge (g(y) \approx h(z,x)))$. Rule 4 of the reduction system is applicable. By applying this rule on $\varphi \wedge (g(y) \approx h(z,x))$ we get $\sigma := \bot$, and hence it should be reduced to $\bot$. Therefore according to the algorithm $\mathtt{Reduce}(\varphi \wedge (g(y) \approx h(z,x))) = (\bot, \emptyset)$. So $\mathtt{GDPLL}(\varphi \wedge (g(y) \approx h(z,x))) = (\mathsf{UNSAT}, \emptyset)$.

Hence according to the `GDPLL` algorithm we have to calculate $\mathtt{GDPLL}(\varphi|_{g(y) \not\approx h(z,x)} \wedge (g(y) \not\approx h(z,x)))$. By Definition 4.7 we have $\varphi|_{g(y) \not\approx h(z,x)} = \{\{f(x) \approx y\}, \{x \not\approx z\}\}$, and so $\varphi|_{g(y) \not\approx h(z,x)} \wedge (g(y) \not\approx h(z,x)) = \{\{f(x) \approx y\}, \{x \not\approx z\}, \{g(y) \not\approx h(z,x)\}\}$. As the first immediate step we use the Reduce algorithm to get a reduced form of this CNF. Rule 4 is the only applicable rule; we derive $\sigma = \{y \mapsto f(x)\}$ and $\{\{x \not\approx z\}, \{g(f(x)) \not\approx h(z,x)\}\}$ after applying it on our CNF. Now rule 3 can be applied on $g(f(x)) \not\approx h(z,x)$ and since $\sigma := mgu(g(f(x)) \approx h(z,x)) = \bot$, this CNF must be reduced to $\{\{x \not\approx z\}\}$. Hence $(\{\{x \not\approx z\}\}, \{y \mapsto f(x)\})$ is the outcome of this algorithm since none of the rules of the reduction system can be applied anymore. Hence $(\varphi, \alpha) = (\{\{x \not\approx z\}\}, \{y \mapsto f(x)\})$. $\bot \notin \varphi$ so we continue with the next step. φ has no purely positive clause obviously, hence we must calculate the $\mathsf{SatCriterion}$ algorithm on φ. Let $Y := \emptyset$, $X := \emptyset$, $d_X := 0$ and $i := 0$. Now $\varphi = \{\{x \not\approx z\}\} \neq \emptyset$ so we enter the loop. Let $C := \{x \not\approx z\}$ and $(x \not\approx z) \in C$. Then $\varphi := \emptyset$ and $d_X := max(d_X, depth(z)) = 0$, moreover $i := i + 1 = 1$, $Y := Y \cup \{x\} = \{x\}$ and $X := X \cup \{(x,i)\} = \{(x,1)\}$. Since $\varphi = \emptyset$, we must leave the loop. Now since $g \in \mathtt{Fun}$, we must immediately apply the **Func** algorithm. Let $\alpha := \emptyset$. Let us choose $g \in \mathtt{Fun}$ and a constant symbol c. Then $F[\,] = g([\,])$, $M := d_X + 1 = 1$ and $\sigma := \{x \mapsto g(c)\} \cup \{z \mapsto c \mid z \neq x\}$. Hence $\alpha := \sigma\alpha = \{x \mapsto g(c), y \mapsto f(g(c))\} \cup \{z \mapsto c \mid z \notin \{x,y\}\}$. So that $\mathbf{Func}(\varphi, X, Y, d_X) = \alpha = \{x \mapsto g(c), y \mapsto f(g(c))\} \cup \{z \mapsto c \mid z \notin \{x,y\}\}$. Therefore the `GDPLL` algorithm returns

$$(\mathsf{SAT}, \{x \mapsto g(c),\ y \mapsto f(g(c))\} \cup \{z \mapsto c \mid z \notin \{x,y\}\})$$

as the outcome. So that the original CNF φ is satisfiable and the interpretation

$$\alpha := \{x \mapsto g(c), y \mapsto f(g(c))\} \cup \{z \mapsto c \mid z \notin \{x, y\}\}$$

satisfies it.

So far two techniques are presented in this thesis, which are essentially different (see Chapter 1). Although their underlying logics are not the same, but they share the equality logic with zero and successor. In the next section we show by an example how either of these two methods, BDD-based approach and DPLL-based one, behave on a formula in their common theory.

4.10 Comparison, OBDD and GDPLL methods

The example below illustrates the different outcomes from the OBDD algorithm of Chapter 3 and from GDPLL algorithm of this chapter, on one input formula.

Example 4.45 *Let $\varphi = (x \approx y \wedge S(y) \approx z) \rightarrow S(x) \approx z$. We demonstrate how either of the OBDD and GDPLL methods investigate the (un)satisfiability of this formula:*

- OBDD algorithm. Let us replace $\approx$ with $=$ as we used in the previous chapters. $\varphi\downarrow = ITE(x = y, ITE(S(y) = z, ITE(S(x) = z, \top, \bot), \top), \top)$. This is not $\bot$ so the next step is to calculate $\mathtt{sort}(\varphi\downarrow)$.

  $$\begin{aligned} \mathtt{sort}(\varphi\downarrow) &= ITE(x = y, \mathtt{sort}(\psi|_{x=y})\downarrow, \mathtt{sort}(\psi|_{x \neq y})\downarrow) \\ &= ITE(x = y, (ITE(S(x) = z, ITE(S(x) = z, \top, \bot), \top))\downarrow, \top) \\ &= ITE(x = y, \top, \top) \end{aligned}$$

  This is in the next step simplified to $\top$. Hence $\mathtt{OBDD}(\varphi)$ is $\top$. Therefore φ is a "tautology".

- GDPLL algorithm. φ is equivalent to the CNF $\{\{x \not\approx y,\ S(x) \not\approx y,\ S(x) \not\approx z\}\}$. Let us call it φ as well. $\mathsf{Reduce}(\varphi)$ is φ itself. Moreover $\bot \notin \varphi$. So we go to the next step, we check whether $\mathsf{SatCriterion}(\varphi)$ holds. It holds, hence $\mathtt{GDPLL}(\varphi)$ is SAT. Therefore the original formula is "satisfiable". With the second GDPLL algorithm (Definition 4.43) we can also compute a witness for the formula.

Using the GDPLL algorithm, it is also possible to find out whether ϕ is a tautology. We could do it by giving the $\neg\phi$ to the algorithm. In this case the output would be UNSAT, which meanes that $\neg\phi$ is not satisfiable, and as a result ϕ is a tautology.

4.11 Conclusion

In this chapter we introduced a generalization of the well-known DPLL algorithm, for ground term algebra. Our so called GDPLL algorithm decides the (un)satisfiability

of CNF formulas in equational logic with constructors. Then we gave yet another extension of the algorithm, through which a witness for satisfiable formulas, can be obtained. One of the crucial parts here was presenting a system by which redundancies are removed and unit clauses are set to true, with their appropriate impact on the other clauses (the whole formula as a result). We succeeded in giving a “proper” reduction system, by which a CNF is reduced, well enough.

As we mentioned in the introduction (Chapter 1), either of OBDD and GDPLL algorithms can outperform the other, at a time, for certain formulas. In the last section, we presented an example which show how either of our two methods operate on a formula. The example we depict there is quite simple and one can easily see that it is a tautology. However in practice it is not always as visible (see [BvdPTZ04b]).

This algorithm is implemented in C, by van de Pol. It is available at
`http://homepages.cwi.nl/~ vdpol/gdpll.html`.
Some of the benchmarks are present in [BvdPTZ04b]. The benchmarks also show that our technique competes with existing encoding methods.

Our GDPLL algorithm, for ground term algebras, here is an instance of the general framework we present in [BvdPTZ04b]. Tveretina [Tve05], also has introduced another instance of this framework. Her algorithm decides the logic of Equality with Uninterpreted Functions.

5

Ground Term Algebra with Recognizers

The algorithm in Chapter 4 works for constructor symbols only (such as zero, successor, nil and cons). Here we extend the framework to recognizer predicates (standard ones are like nil?, succ?, cons?, zero?). We show how these predicates can be eliminated by introducing new variables.

5.1 Introduction

In the PVS theorem prover (see Chapters 1 and 7) datatypes are declared by providing a set of constructors, together with their associated *recognizers* and accessors (destructors). When a datatype is type-checked a new theory will be automatically created. This new theory provides the necessary axioms to ensure that the datatype builds the algebra defined by the constructors. For datatypes with several constructors such an axiom could grow very large, slowing down the proof process. Proving new theorems, based on these axioms, can sometimes be done by PVS automatically. More often, they will need the user's hint to be proved. So, having a proper decision procedure which would prove a larger family of these generated axioms, automatically, can save effort and time.

Furthermore, many programming languages also support recursively defined data structures. The best known example is *LISP* list structure, with constructors, destructors and recognizers.

Recognizers are unary predicates of the form $f?$, $g?$,.... They operate as subtypes of the datatype they belong to. We add these predicates to the language of ground term algebra (Chapter 4). The new structure based on this signature will be called *ground term algebra with recognizers.* Then we will discuss the problem of reasoning about the formulas in a ground term algebra with recognizers. Our approach is based on reusing the results of Chapter 4 to propose a decision procedure for the extension

of ground term algebras with recognizers.

For instance the theory of LISP list structure includes constructors *nil* and *cons* and their corresponding recognizers *nil?* and *cons?*. A simple example is that with our new decision procedure we will be able to determine directly, the (un)satisfiability of formulas like

$$(nil?(t) \vee cons?(t))$$

which says that every term t in LISP is either in the range of $nil[\,]$ or in the range of $cons[\,]$. In other words, t is either of the form nil or $cons(s, w)$ for a pair of terms s and w. Using our decision procedure we can show that the formula above is a tautology.

The pattern we follow to decide about the satisfiability of formulas in Conjunctive Normal Form (i.e. CNF) is as follows: First we transform the formula with recognizers to some formula in the ground term algebra (without recognizers); we call this process *transformation*. Then we apply the so-called `GDPLL` algorithm (see Definition 4.27) on the newly produced formula in order to determine whether it is satisfiable.

Here is how this chapter is structured: We dedicate Section 5.2 to some preliminary definitions. Also we define the new structure, ground term algebra with recognizers. In Section 5.3 we present an algorithm to transform recognizers to the language of ground term algebra. Then using that transformation we introduce a decision procedure for ground term algebra with recognizers. We prove the completeness of this decision procedure in Section 5.4. The theory of Lists is a good example to apply our decision procedure on; we do this in Section 5.5.

5.2 Recognizers

Here we introduce the extension of ground term algebra with recognizers. First, we explain what a recognizer is and we introduce the kind of signature and formulas that we will work with in this chapter. Then, we identify the theory itself, and we will give the theory of natural numbers $\mathbb{N}$ with recognizers as an example.

5.2.1 Syntax

Let $\texttt{Fun} = \{f, g, h, \dots\}$ be a set of function symbols. In this chapter $\texttt{Fun}$ has a finite number of elements. We associate with $\texttt{Fun}$ the set $\{f?, g?, h?, \dots\}$ of unary predicate symbols, called recognizers. We name this set $\texttt{Rec}_{\texttt{Fun}}$. Hence

$$\texttt{Rec}_{\texttt{Fun}} = \{f? \mid f \in \texttt{Fun}\}.$$

In this chapter we work with signatures of the form $\Sigma = (\texttt{Fun}, \{\approx\} \cup \texttt{Rec}_{\texttt{Fun}})$, where $\{\approx\} \cup \texttt{Rec}_{\texttt{Fun}}$ is the set of predicate symbols $\texttt{Pr}$.

We have some preliminary definitions for this language which are similar to those in Chapter 4. The set $\texttt{Term}(\Sigma, \texttt{Var})$ of *terms* over the signature Σ is defined identical to that of previous chapter. Atoms are of the form $t \approx s$ or $f?(t)$ for some terms t and s, or a recognizer $f?$. we represent the set of atoms by $\texttt{At}$. Literals are atoms or negation of atoms. $\texttt{Lit}$ is the set of all literals. Clauses are finite sets of literals (expressing the disjunction of those literals). We work with *conjunctive normal form* (CNF) formulas. Each CNF is represented by a finite set of clauses.

Here is an example of a formula in CNF that involves recognizers:

$$\phi \ = \ \{\{z \not\approx f(x,y)\},\ \{\neg g?(t),\ t \approx s\}\}.$$

Below we identify some other notations which are frequently used in this chapter.

Definition 5.1 *Given a CNF ϕ we define:*

- $\texttt{Lit}_{\texttt{rec}}(\phi) = \{l \mid l \in \texttt{Lit}(\phi),\ l = f?(t) \text{ or } l = \neg f?(t)\}$
  The set of the literals of ϕ that involve a recognizer.
- $\texttt{Lit}_{\texttt{recp}}(\phi) = \texttt{Lit}_{\texttt{rec}}(\phi) \ \cap \ \texttt{Plit}(\phi)$
  The set of all positive literals in $\texttt{Lit}_{\texttt{rec}}(\phi)$.
- $\texttt{Lit}_{\texttt{recn}}(\phi) = \texttt{Lit}_{\texttt{rec}}(\phi) \ \cap \ \texttt{Nlit}(\phi)$
  The set of all negative literals in $\texttt{Lit}_{\texttt{rec}}(\phi)$.

Example 5.2 *If $\phi = \{\{\neg f?(t)\}, \{x \approx y, g?(s)\}\}$ then*

- $\texttt{Lit}_{\texttt{rec}}(\phi) = \{\neg f?(t), g?(s)\}$
- $\texttt{Lit}_{\texttt{recp}}(\phi) = \{g?(s)\}$
- $\texttt{Lit}_{\texttt{recn}}(\phi) = \{\neg f?(t)\}$.

5.2.2 Semantics

Given a signature Σ of the above mentioned form, we define a structure called *ground term algebra with recognizers* over Σ. The idea is that $\approx$ again represents equality and that terms are only interpreted by ground terms i.e. they belong to $\texttt{Term}(\Sigma)$.

All the other notations which were introduced in Chapter 4 will have a similar meaning also in this chapter.

Definition 5.3 (ground term algebra with recognizers)
Suppose $\Sigma = (\texttt{Fun}, \{\approx\} \cup \texttt{Rec}_{\texttt{Fun}})$ is a signature. We define a structure over this signature called ground term algebra with recognizers *as follows.*

- $D = \texttt{Term}(\Sigma)$ *is the set of terms,*

- *$f_D(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$ for all $f \in \texttt{Fun}$ and all $t_1, \ldots, t_n \in \texttt{Term}(\Sigma)$, where n is the arity of f,*
- *$\approx_D$ is syntactic equality on terms,*
- *$f?_D : D \rightarrow \{\texttt{true}, \texttt{false}\}$ is a unary predicate (called recognizer) with the following property:*

$$f?_D(t) = \begin{cases} \texttt{true} & \text{if } t = f(s_1, \ldots, s_n) \text{ for some } s_1, \ldots, s_n \in D \\ \texttt{false} & \text{otherwise.} \end{cases}$$

Later on we will drop the subscript D from f_D, $\approx_D$ and $f?_D$, for simplicity.

Example 5.4 *The theory of natural numbers over the signature $\Sigma = (\{0, Succ\}, \{\approx, 0?, Succ?\})$ is a ground term algebra with recognizers, with*

- *$D = \mathbb{N} = \texttt{Term}(\Sigma) = \{0, Succ(0), Succ(Succ(0)), \ldots\}$,*
- *$0_D = 0$ and $Succ_D(t) = Succ(t)$,*
- *$\approx_D$ is $\approx$ on natural numbers,*
- $$0?_D(t) = \begin{cases} \texttt{true} & \text{if } t = 0 \\ \texttt{false} & \text{otherwise,} \end{cases}$$

  and

  $$Succ?_D(t) = \begin{cases} \texttt{true} & \text{if } t = Succ(s) \text{ for some } s \in \mathbb{N} \\ \texttt{false} & \text{if } t = 0. \end{cases}$$

5.3 Transformation

The purpose of this section is to show how to transform all the formulas of ground term algebra with recognizers to equivalent ones in ground term algebra. In other words we want to replace the CNF formulas which include recognizers to equivalent recognizer free CNFs. First we will present the technique for positive literals which involve recognizers, and then we will propose a similar method for the negative ones.

5.3.1 Positive literals

Definition 5.5 *Let l be a positive literal of the form $f?(t)$, with f of arity n. Let $x_1, \ldots, x_n \in \texttt{Var}$ be n fresh variables (i.e. they do not occur in ϕ). We define the following two notations:*

- $\mathtt{tr}_{x_1,\ldots,x_n}(l) = (t \approx f(x_1,\ldots,x_n))$
- $\phi[l := \mathtt{tr}_{x_1,\ldots,x_n}(l)]$ *is the formula obtained by replacing all occurrences of* l *in* ϕ *with* $\mathtt{tr}_{x_1,\ldots,x_n}(l)$.

Example 5.6 *If* $\phi = \{\{succ?(t)\},\{t \approx 0\}\}$ *then* $\mathtt{tr}_y(succ?(t))$ *is* $t \approx succ(y)$ *and* $\phi[succ?(t) := (t \approx succ(y))]$ *is* $\{\{t \approx succ(y)\},\{t \approx 0\}\}$. *This CNF is obviously unsatisfiable.*

The following lemma shows that the replacement defined in Definition 5.5 is correct (i.e. it preserves satisfiability).

Lemma 5.7 *Let* ϕ *be a CNF, let* $f?(t) \in \mathtt{Lit}_{\mathtt{recp}}(\phi)$ *with* $arity(f) = n$, *and let* $x_1,\ldots,x_n$ *be a sequence of distinct variables such that* $x_i \notin \mathtt{var}(\phi)$ *for* $1 \le i \le n$. *Then* ϕ *is satisfiable if and only if* $\phi[f?(t) := \mathtt{tr}_{x_1,\ldots,x_n}(f?(t))]$ *is satisfiable.*

Proof. Suppose $\phi = \{C_1, \ldots, C_n\}$. Let $\mathtt{tr}(C_i)$ represent the clause of $\phi[f?(t) := \mathtt{tr}_{x_1,\ldots,x_n}(f?(t))]$ corresponding to C_i, for all $1 \le i \le n$.

- Suppose ϕ is satisfiable.
  Then there is an interpretation $\sigma : \mathtt{Var} \to D$ such that $C_i^\sigma = \mathtt{true}$ for all $1 \le i \le n$. We prove that there is an interpretation $\bar\sigma : \mathtt{Var} \to D$ such that $(\mathtt{tr}(C_i))^{\bar\sigma} = \mathtt{true}$, for all $1 \le i \le n$.
  - If $(f?(t))^\sigma = \mathtt{true}$, then $f?(t^\sigma) = \mathtt{true}$, so $\Rightarrow t^\sigma = f(s_1,\ldots,s_n)$ for some $s_1,\ldots,s_n \in D$.
    We define the interpretation $\bar\sigma : \mathtt{Var} \to D$ as follows:
    $$y^{\bar\sigma} = \begin{cases} s_i & \text{if } y = x_i \\ y^\sigma & \text{otherwise.} \end{cases}$$
    According to our assumption, the variables $x_1,\ldots,x_n$ are not occurring in ϕ. To show that $(\mathtt{tr}(C_i))^{\bar\sigma} = \mathtt{true}$ we distinguish two cases
    * Suppose $f?(t) \notin C_i$. Then $\mathtt{tr}(C_i) = C_i$, by definition. Therefore $(\mathtt{tr}(C_i))^{\bar\sigma} = C_i^{\bar\sigma} = C_i^\sigma = \mathtt{true}$, since $x_1,\ldots,x_n$ are not occurring in ϕ.
    * If $f?(t) \in C_i$, then $t^{\bar\sigma} = t^\sigma = f(s_1,\ldots,s_n) = f((x_1)^{\bar\sigma},\ldots,(x_n)^{\bar\sigma}) = f(x_1,\ldots,x_n))^{\bar\sigma}$ so $\mathtt{tr}_{x_1,\ldots,x_n}(f?(t)))^{\bar\sigma} = (t \approx f(x_1,\ldots,x_n))^{\bar\sigma} = \mathtt{true}$. Since $\mathtt{tr}_{x_1,\ldots,x_n}(f?(t)) \in \mathtt{tr}(C_i)$, it follows that $(\mathtt{tr}(C_i))^{\bar\sigma} = \mathtt{true}$.
  - If $(f?(t))^\sigma = \mathtt{false}$, then since $C_i^\sigma = \mathtt{true}$ for all $1 \le i \le n$, for every C_i there is a literal $l_i \in C_i$ such that $l_i \ne f?(t)$ and $l_i^\sigma = \mathtt{true}$. Since $l_i \ne f?(t)$, it follows that $l_i \in \mathtt{tr}(C_i)$ and hence $(\mathtt{tr}(C_i))^\sigma = \mathtt{true}$. So we can take $\bar\sigma = \sigma$.

We conclude that $\phi[f?(t) := \texttt{tr}_{x_1,\ldots,x_n}(f?(t))]$ is satisfiable.

- Suppose $\phi[f?(t) := \texttt{tr}_{x_1,\ldots,x_n}(f?(t))]$ is satisfiable. Then there is an interpretation $\sigma : \texttt{Var} \to D$ such that $(\texttt{tr}(C_i))^\sigma = \texttt{true}$ for all $1 \leq i \leq n$. We show that $C_i^\sigma = \texttt{true}$ for all $1 \leq i \leq n$.
  - If $f?(t) \notin C_i$, then $C_i = \texttt{tr}(C_i)$, by definition. Therefore $C_i^\sigma = (\texttt{tr}(C_i))^\sigma = \texttt{true}$.
  - If $f?(t) \in C_i$, then $\texttt{tr}(C_i) = (C_i - \{f?(t)\}) \cup \{t \approx f(x_1, \ldots, x_n)\}$.
    * If $(f?(t))^\sigma = \texttt{true}$, then $C_i^\sigma = \texttt{true}$, since $f?(t) \in C_i$.
    * If $(f?(t))^\sigma = \texttt{false}$, then $f?(t^\sigma) = (f?(t))^\sigma = \texttt{false}$. Then for all $s_1, \ldots, s_n \in D$ $t^\sigma \neq f(s_1, \ldots, s_n)$. So in particular $(t \approx f(x_1, \ldots, x_n))^\sigma = \texttt{false}$.
      Since $(\texttt{tr}(C_i))^\sigma = \texttt{true}$, there exists an $l \in \texttt{tr}(C_i)$, $l \neq f?(t)$ such that $l^\sigma = \texttt{true}$. Hence $l \in C_i$ by definition, and $C_i^\sigma = \texttt{true}$ too.

  We conclude that ϕ is satisfiable.

■

5.3.2 Negative literals

In this section we will transform the negative literals involving recognizers to equivalent ones without recognizers. Here we cannot simply use the $\neg\texttt{tr}(l)$ as a transformation for $\neg l$, because these two are not equivalent. We demonstrate this with the following example.

Example 5.8 *Let $l = f?(f(t))$. Then according to Definition 5.3 $\neg l = \texttt{false}$.*
But on the other hand $\texttt{tr}(l) = f(x) \approx f(t)$ and so $\neg\texttt{tr}(l) = f(x) \not\approx f(t)$, which is $\texttt{true}$ *if e.g. $x \neq t$ and is* $\texttt{false}$ *if $x = t$. Hence replacing $\neg\texttt{tr}(l)$ by $\neg l$ would not preserve (un)satisfiability.*

Definition 5.9 *Let l be a negative literal involving a recognizer, say $l = \neg f?(t)$. We define two new notations, as below:*

- $\texttt{tr}(l) = \{g?(t) \mid g \in \texttt{Fun},\ g \neq f\}$
- $\phi\|_{\texttt{tr}(l)} = \{C \mid C \in \phi,\ l \notin C\} \ \cup\ \{(C - \{l\}) \cup \texttt{tr}(l) \mid C \in \phi,\ l \in C\}$
  A special substitution that replaces a negative recognizer by a suitable set of positive recognizers.

Example 5.10 *The CNF* $\phi = \{\{\neg f?(f(t))\}\}$ *is unsatisfiable, regarding Definition 5.3. Now let* $l = \neg f?(f(t))$. *Then* $\texttt{tr}(l) = \{g?(f(t)) \mid g \in \texttt{Fun},\ g \neq f\}$ *and so* $\phi\|_{\texttt{tr}(l)} = \{\{g?(f(t)) \mid g \in \texttt{Fun},\ g \neq f\}\}$, *which is unsatisfiable according to Definition 5.3.*

Example 5.11 *If* $\phi = \{\{f?(t)\}, \{x \approx y, \neg f?(t)\}\}$ *in a signature with* $\texttt{Fun} = \{g, f\}$ *then:*

$$\begin{aligned}
\phi\|_{\texttt{tr}(\neg f?(t))} &= \{\{f?(t)\}\} \ \cup\ \{\{x \approx y\} \ \cup\ \texttt{tr}(\neg f?(t))\} \\
&= \{\{f?(t)\}\} \ \cup\ \{\{x \approx y\} \ \cup\ \{h?(t) \mid h \in \texttt{Fun},\ h \neq f\}\} \\
&= \{\{f?(t)\},\ \{x \approx y, g?(t)\}\}.
\end{aligned}$$

The following lemma shows that the replacement above preserves satisfiability.

Lemma 5.12 *Let* ϕ *be a CNF and* $\neg f?(t) \in \texttt{Lit}_{\texttt{recn}}(\phi)$. *Then* ϕ *is satisfiable if and only if* $\phi\|_{\texttt{tr}(\neg f?(t))}$ *is satisfiable.*

Proof. Suppose $\phi = \{C_1,\ \ldots, C_n\}$. Then $\phi\|_{\texttt{tr}(\neg f?(t))} = \{\texttt{tr}(C_1), \ldots, \texttt{tr}(C_n)\}$, where $\texttt{tr}(C_i) = C_i$ if $\neg f?(t) \notin C_i$ and
$\texttt{tr}(C_i) = (C_i - \{\neg f?(t)\}) \cup \texttt{tr}(\neg f?(t))$ if $\neg f?(t) \in C_i$, for $1 \leq i \leq n$.

- Suppose ϕ is satisfiable. Then there is an interpretation $\sigma : \texttt{Var} \to D$ such that $C_i^\sigma = \texttt{true}$ for all $1 \leq i \leq n$.
  We prove that there is an interpretation $\bar{\sigma} : \texttt{Var} \to D$ such that $(\texttt{tr}(C_i))^{\bar{\sigma}} = \texttt{true}$, for all $1 \leq i \leq n$. We show that $\bar{\sigma} = \sigma$ works here.
  - If $\neg f?(t) \notin C_i$, then $\texttt{tr}(C_i) = C_i$, and hence $(\texttt{tr}(C_i))^\sigma = C_i^\sigma = \texttt{true}$.
  - If $\neg f?(t) \in C_i$, then $\texttt{tr}(C_i) = (C_i - \{\neg f?(t)\}) \cup \texttt{tr}(\neg f?(t))$.
    * If $(\neg f?(t))^\sigma = \texttt{true}$, then $f?(t^\sigma) = (f?(t))^\sigma = \texttt{false}$, so $t^\sigma \neq f(s_1, \ldots, s_n)$, for all $s_1, \ldots, s_n \in D$. Therefore, since $t^\sigma \in D = \texttt{Term}(\Sigma)$, there are a function symbol $g \neq f$ and m terms $r_1, \ldots, r_m \in D$ such that $t^\sigma = g(r_1, \ldots, r_m)$.
      Hence $(g?(t))^\sigma = g?(t^\sigma) = \texttt{true}$. Moreover $g?(t) \in \texttt{tr}(\neg f?(t)) \subseteq \texttt{tr}(C_i)$, so $(\texttt{tr}(C_i))^\sigma = \texttt{true}$.
    * If $(\neg f?(t))^\sigma = \texttt{false}$, then since $C_i^\sigma = \texttt{true}$, there exists a literal $l_k \neq \neg f?(t)$, $1 \leq k \leq m_i$ such that $l_k^\sigma = \texttt{true}$. Moreover $l_k \in \texttt{tr}(C_i)$ by definition. Therefore $(\texttt{tr}(C_i))^\sigma = \texttt{true}$.

  We conclude that $\phi\|_{\texttt{tr}(\neg f?(t))}$ is satisfiable.

- Suppose $\phi\|_{\mathtt{tr}(\neg f?(t))}$ is satisfiable. Then there is an interpretation $\sigma : \mathtt{Var} \to D$ such that $(\mathtt{tr}(C_i))^\sigma = \mathtt{true}$ for all $1 \leq i \leq n$.
  We show that $C_i^\sigma = \mathtt{true}$ for all $1 \leq i \leq n$.
  - If $\neg f?(t) \notin C_i$, then $C_i = \mathtt{tr}(C_i)$. Therefore $C_i^\sigma = (\mathtt{tr}(C_i))^\sigma = \mathtt{true}$.
  - If $\neg f?(t) \in C_i$, then $\mathtt{tr}(C_i) = (C_i - \{\neg f?(t)\}) \cup \mathtt{tr}(\neg f?(t))$.
    * If $(\neg f?(t))^\sigma = \mathtt{true}$, then $C_i^\sigma = \mathtt{true}$, since $\neg f?(t) \in C_i$.
    * If $(\neg f?(t))^\sigma = \mathtt{false}$, then $f?(t^\sigma) = (f?(t))^\sigma = \mathtt{true}$, so $t^\sigma = f(s_1, \ldots, s_n)$ for some $s_1, \ldots, s_n \in D$. Hence for all function symbols $g \in \mathtt{tr}(\neg f?(t))$, $(g?(t))^\sigma = g?(t^\sigma) = g?(f(s_1, \ldots, s_n)) = \mathtt{false}$, since $g \neq f$.
      $(\mathtt{tr}(C_i))^\sigma = \mathtt{true}$, therefore there exists an $l \in \mathtt{tr}(C_i)$ such that $l \notin \mathtt{tr}(\neg f?(t))$ and $l^\sigma = \mathtt{true}$.
      $l \in C_i$, because $l \notin \mathtt{tr}(\neg f?(t))$. Therefore $C_i^\sigma = \mathtt{true}$.

  We conclude that ϕ is satisfiable.

■

5.3.3 Algorithm

In this section we are going to present an algorithm to transform formulas with recognizers to formulas without recognizers, and we will subsequently establish that the algorithm is correct.

Definition 5.13 *We present the transformation procedure. The input of this algorithm is a CNF, possibly including recognizers, and the output is a new CNF formula which has no recognizers.*

TR(ϕ)
  begin
    while $\mathtt{Lit}_{\mathtt{recn}}(\phi) \neq \emptyset$ **do**
      choose $l \in \mathtt{Lit}_{\mathtt{recn}}(\phi)$ *;*
      $\phi := \phi\|_{\mathtt{tr}(l)}$ *;*
    od

    while $\mathtt{Lit}_{\mathtt{recp}}(\phi) \neq \emptyset$ **do**
      choose $l \in \mathtt{Lit}_{\mathtt{recp}}(\phi)$ *;*
      let $l = f?(t)$ *;*
      let $n = arity(f)$ *;*

choose $x_1, \ldots, x_n \in (\mathtt{Var} - \mathtt{Var}(\phi))$ *;*
$\phi := \phi[l := \mathtt{tr}_{x_1,\ldots,x_n}(l)]$ *;*
od

return ϕ *;*
end

Example 5.14 *Let* $\phi = \{\{\neg f?(y)\}, \{\neg g?(y)\}\}$ *be a CNF formula in a signature with only two function symbols* f *and* g *(i.e.* $\mathtt{Fun} = \{f, g\}$*). Then* $\mathtt{TR}(\phi)$ *is a formula with no recognizers.*

We first calculate $\mathtt{tr}(\neg f?(y))$ *and* $\mathtt{tr}(\neg g?(y))$ *which will be used in this application of the* TR *algorithm. According to Definition 5.9* $\mathtt{tr}(\neg f?(y)) = \{g?(y)\}$ *and* $\mathtt{tr}(\neg g?(y)) = \{f?(y)\}$.

Now regarding the TR *algorithm, since* $\mathtt{Lit}_{\mathtt{recn}}(\phi) = \{\{\neg f?(y)\}, \{\neg g?(y)\}\} \neq \emptyset$*, we must first go into the first loop.*

Let us choose $l = \neg f?(y)$*. Then* $\phi := \phi\|_{\mathtt{tr}(\neg f?(y))} = \{\{\neg g?(y)\}, \{g?(y)\}\}$ *regarding Definition 5.1.*

Once more $\mathtt{Lit}_{\mathtt{recn}}(\phi) = \{\{\neg g?(y)\}\} \neq \emptyset$*. Now* $l = \neg g?(y)$ *is the only choice we can have in* $\mathtt{Lit}_{\mathtt{recn}}(\phi)$*. Hence* $\phi := \phi\|_{\mathtt{tr}(\neg g?(y))} = \{\{g?(y)\}, \{f?(y)\}\}$.

This time $\mathtt{Lit}_{\mathtt{recn}}(\phi) = \emptyset$*, therefore we leave the first loop and go to the next where the condition to get in the loop is* $\mathtt{Lit}_{\mathtt{recp}}(\phi) \neq \emptyset$.

$\mathtt{Lit}_{\mathtt{recp}}(\phi) = \{\{g?(y)\}, \{f?(y)\}\} \neq \emptyset$.
Let $l = f?(y)$ *and* $n = arity(f)$*. Let* $x_1, \ldots, x_n \in (\mathtt{Var} - \{y\})$*. Hence*
$\phi := \phi[f?(y) := \mathtt{tr}_{x_1,\ldots,x_n}(f?(y))] = \{\{g?(y)\}, \{y \approx f(x_1, \ldots, x_n)\}\}$ *regarding Definition 5.1.*

We check the condition once more.
$\mathtt{Lit}_{\mathtt{recp}}(\phi) = \{\{g?(y)\}\} \neq \emptyset$*. Here the only choice for* l *is* $l = g?(y)$.
Let $m = arity(g)$ *and* $z_1, \ldots, z_m \in (\mathtt{Var} - \{y, x_1, \ldots, x_n\})$*. Hence*
$\phi := \phi[g?(y) := \mathtt{tr}_{z_1,\ldots,z_m}(g?(y))] = \{\{y \approx g(z_1, \ldots, z_m)\}, \{y \approx f(x_1, \ldots, x_n)\}\}$.

For this new ϕ *we have* $\mathtt{Lit}_{\mathtt{recp}}(\phi) = \emptyset$*. So we must leave the second loop and therefore the algorithm ends where* ϕ *has the value:*
$\{\{y \approx g(z_1, \ldots, z_m)\}, \{y \approx f(x_1, \ldots, x_n)\}\}$.

The next two lemmas will help us to prove termination of our algorithm.

Lemma 5.15 *Let* $l \in \mathtt{Lit}_{\mathtt{recn}}(\phi)$*. Then* $\|\mathtt{Lit}_{\mathtt{recn}}(\phi\|_{\mathtt{tr}(l)})\| < \|\mathtt{Lit}_{\mathtt{recn}}(\phi)\|$.

Proof. According to Definition 5.9, $\mathtt{tr}(l)$ does not include any negative literal. Hence according to Definition 5.1 $\mathtt{Lit}_{\mathtt{recn}}(\phi\|_{\mathtt{tr}(l)}) \subseteq \mathtt{Lit}_{\mathtt{recn}}(\phi)$. Also $l \notin \mathtt{Lit}_{\mathtt{recn}}(\phi\|_{\mathtt{tr}(l)})$ by Definition 5.9, hence $l \in \mathtt{Lit}_{\mathtt{recn}}(\phi) - \mathtt{Lit}_{\mathtt{recn}}(\phi\|_{\mathtt{tr}(l)})$. Therefore $\mathtt{Lit}_{\mathtt{recn}}(\phi\|_{\mathtt{tr}(l)}) \subset \mathtt{Lit}_{\mathtt{recn}}(\phi)$, and so $\|\mathtt{Lit}_{\mathtt{recn}}(\phi\|_{\mathtt{tr}(l)})\| < \|\mathtt{Lit}_{\mathtt{recn}}(\phi)\|$. ∎

Lemma 5.16 *Let* $l \in \mathtt{Lit}_{\mathtt{recp}}(\phi)$. *Then* $\|\mathtt{Lit}_{\mathtt{recp}}(\phi[l := \mathtt{tr}_{x_1,\dots,x_m}(l)])\| < \|\mathtt{Lit}_{\mathtt{recp}}(\phi)\|$.

Proof. Regarding Definition 5.5, $\mathtt{tr}_{x_1,\dots,x_m}(l)$ does not involve any recognizers. Hence $\mathtt{Lit}_{\mathtt{recp}}(\phi[l := \mathtt{tr}_{x_1,\dots,x_m}(l)]) \subseteq \mathtt{Lit}_{\mathtt{recp}}(\phi)$. Moreover $l \in \mathtt{Lit}_{\mathtt{recp}}(\phi) - \mathtt{Lit}_{\mathtt{recp}}(\phi[l := \mathtt{tr}_{x_1,\dots,x_m}(l)])$. Therefore $\mathtt{Lit}_{\mathtt{recp}}(\phi[l := \mathtt{tr}_{x_1,\dots,x_m}(l)]) \subset \mathtt{Lit}_{\mathtt{recp}}(\phi)$, and so $\|\mathtt{Lit}_{\mathtt{recp}}(\phi[l := \mathtt{tr}_{x_1,\dots,x_m}(l)])\| < \|\mathtt{Lit}_{\mathtt{recp}}(\phi)\|$. ∎

The next theorem shows that the algorithm terminates. Intuitively, first we replace the negative recognizers with appropriate positive ones and when no negative recognizer exist then we replace all positive recognizers with proper literals, not including recognizers. This way after finite steps there will be no recognizer left, moreover all these steps preserve (un)satisfiability.

Theorem 5.17 (Termination) *The* TR *algorithm is terminating and the result is a formula without recognizers.*

Proof. The first **while-**loop in the algorithm in Definition 5.13 is terminating, since by Lemma 5.15 the size of $\mathtt{Lit}_{\mathtt{recn}}(\phi)$ reduces in each iteration. The second **while-**loop in that algorithm is terminating too, since by Lemma 5.16, the size of $\mathtt{Lit}_{\mathtt{recp}}(\phi)$ is reducing.
The algorithm ends when $\mathtt{Lit}_{\mathtt{recn}}(\phi) = \mathtt{Lit}_{\mathtt{recp}}(\phi) = \emptyset$. Therefore $\mathtt{Lit}_{\mathtt{rec}}(\phi) = \mathtt{Lit}_{\mathtt{recn}}(\phi) \cup \mathtt{Lit}_{\mathtt{recp}}(\phi) = \emptyset$ and hence ϕ contains no recognizers. In other words ϕ is in the corresponding ground term algebra of the assumed structure. ∎

Below we show that the algorithm is correct, in other words, it preserves satisfiability.

Theorem 5.18 (Sat Criterion) *A CNF* ϕ *is satisfiable if and only if* $\mathtt{TR}(\phi)$ *is satisfiable.*

Proof. According to Theorem 5.17 $\mathtt{TR}(\phi)$ always exists. Now regarding Lemma 5.12, ϕ is satisfiable if and only if the outcome of the first **while-**loop in the TR algorithm is satisfiable. And regarding Lemma 5.7, this is satisfiable if and only if the outcome of the second **while-**loop in that algorithm is satisfiable. Therefore ϕ is satisfiable if and only if the outcome of the whole algorithm, which is $\mathtt{TR}(\phi)$, is satisfiable.

∎

5.4 Decision Procedure

We now combine $\texttt{TR}(\phi)$ and the algorithm of Chapter 4.

Definition 5.19 *Given a formula ϕ we define the satisfiability checking procedure* $\texttt{RGDPLL}(\phi)$ *as below:*

$$\texttt{RGDPLL}(\phi) := \texttt{GDPLL}(\texttt{TR}(\phi))$$

Theorem 5.20 (Soundness and Completeness) *Let the CNF ϕ be a formula which possibly includes some recognizers. Then the following properties hold:*

- *ϕ is satisfiable iff* $\texttt{RGDPLL}(\phi) = \mathsf{SAT}$.
- *ϕ is unsatisfiable iff* $\texttt{RGDPLL}(\phi) = \mathsf{UNSAT}$.

Proof.

- By Theorem 5.18 ϕ is satisfiable iff $\texttt{TR}(\phi)$ is satisfiable. $\texttt{TR}(\phi)$ is satisfiable iff $\texttt{GDPLL}(\texttt{TR}(\phi)) = \mathsf{SAT}$, regarding Theorem 4.39. Hence regarding Definition 5.19, ϕ is satisfiable iff $\texttt{RGDPLL}(\phi) = \mathsf{SAT}$.

- This can be derived in a similar fashion as above.

■

Example 5.21 $\phi = \{\{\neg f?(y)\}, \{\neg g?(y)\}\}$ *is unsatisfiable if* $\texttt{Fun} = \{f, g\}$ (*i.e. there are only two distinct function symbols in the signature*).

According to *Soundness and Completeness* theorem (Theorem 5.20) we only need to show that $\texttt{RGDPLL}(\phi) = \mathsf{UNSAT}$. In other words we need to prove that $\texttt{GDPLL}(\texttt{TR}(\phi)) = \mathsf{UNSAT}$.
Using Example 5.14 we know that $\texttt{TR}(\phi) = \{\{y \approx g(z_1, \ldots, z_m)\}, \{y \approx f(x_1, \ldots, x_n)\}\}$. We apply the $\texttt{GDPLL}$ algorithm on this formula which includes no recognizers.

According to the $\texttt{GDPLL}$ algorithm (Definition 4.27) we first need to reduce $\texttt{TR}(\phi)$.

$\mathsf{mgu}(\{\{y \approx g(z_1, \ldots, z_m)\}, \{y \approx f(x_1, \ldots, x_n)\}\}) = \emptyset$, therefore according to Definition 4.20 rule 4, $\texttt{Reduce}(\{\{y \approx g(z_1, \ldots, z_m)\}, \{y \approx f(x_1, \ldots, x_n)\}\}) = \{\bot\}$.

Hence $\texttt{GDPLL}(\texttt{TR}(\phi))$ returns UNSAT.

Therefore $\texttt{RGDPLL}(\phi) = \texttt{GDPLL}(\texttt{TR}(\phi)) = \mathsf{UNSAT}$.

5.5 Lists and List Operations

We give here another example of a ground term algebra with recognizers. It is the list structure of the programming language LISP ([McC62]).

Definition 5.22 *Suppose* $\Sigma = (\{\underline{nil}, \underline{cons}\}, \{\approx, \underline{nil}?, \underline{cons}?\})$ *is a signature, such that* $\underline{nil}$ *is of arity zero and* $\underline{cons}$ *is of arity two. We define lists over* Σ *as below:*

- $D = List = \mathtt{Term}(\Sigma)$ *is the set of all lists, identified below:*

$$List ::= \underline{nil} \mid \underline{cons}(List, List)$$

- *the constructors* $nil :\rightarrow \ List$ *and* $cons : List \times List \rightarrow \ List$ *are the functions in this structure, in which*

$$\begin{aligned} nil &:= \underline{nil} \\ cons(l, l') &:= \underline{cons}(l, l') \end{aligned}$$

- $\approx$ *is the syntactical equality on List.*

- $nil?$ *and* $cons?$ *represent the recognizers, for which the two following properties hold:*

$$nil?(t) = \begin{cases} \mathtt{true} & \text{if } t = nil \\ \mathtt{false} & \text{otherwise,} \end{cases}$$

  and

$$cons?(t) = \begin{cases} \mathtt{true} & \text{if } t = cons(s, w) \text{ for some } s, w \in List \\ \mathtt{false} & \text{if } t = nil. \end{cases}$$

Example 5.23 *Here are some terms in this structure:*

$\underline{nil}$
$\underline{cons}(\underline{nil},\ \underline{nil})$
$\underline{cons}(\underline{nil},\ \underline{cons}(\underline{nil},\ \underline{nil}))$
$\underline{cons}(\underline{cons}(\underline{nil},\ \underline{nil}),\ \underline{cons}(\underline{nil},\ \underline{nil}))$

Below there are two examples where we apply our decision procedure to formulas on list structures.

Example 5.24 $\phi = \{\{nil?(x), cons?(x)\}\}$ *is a tautology.*

To show that ϕ is a tautology, we only need to prove that its negation $\neg\phi$ is unsatisfiable. Using ϕ's CNF and DeMorgan's rules we get that
$\neg\phi = \{\{\neg nil?(x)\}, \{\neg cons?(x)\}\}$. LISP has only two function symbols *nil* and *cons*, hence regarding Example 5.21, $\neg\phi$ is unsatisfiable. Therefore ϕ itself is a tautology.

Consider a formula like $\phi := \forall x, y : \; cons(u, v) \not\approx cons(x, y)$. We can not express ϕ in ground term algebra as $cons(u, v) \not\approx cons(x, y)$ since $\texttt{GDPLL}(cons(u, v) \not\approx cons(x, y))$ is SAT, while ϕ is unsatisfiable. Using recognizers we can express it by an equivalent formula $\psi = \neg cons?(cons(u, v))$. Below we show that ψ is unsatisfiable.

Example 5.25 $\psi = \{\{\neg cons?(cons(u, v))\}\}$ *is unsatisfiable.*
$\texttt{tr}(\neg cons?(cons(u, v)))) = \{nil?(cons(u, v)))\}$ so
$\psi\|_{\texttt{tr}(\neg cons?(cons(u,v)))} = \{\{nil?(cons(u, v)))\}\}$.
Also $\texttt{tr}(nil?(cons(u, v))) = \{nil \approx cons(u, v)\}$, hence $\texttt{TR}(\psi) = \{\{nil \approx cons(u, v)\}\}$ which is unsatisfiable.

5.6 Conclusion

In this chapter we extended the decision procedure of Chapter 4 to the theory of ground term algebras with recognizers. Recognizers are mostly used in theorem provers, for declaration of datatypes. Moreover in the LISP programming language, they are used in list structures.

Our method is based on transforming a formula with recognizers to one without recognizers. We do this in a way that the formula in the end includes possibly some few (finite number) more literals. This way the growth factor is linear. The CNF obtained after transformations can then be decided by the GDPLL algorithm. Adding destructors also to the theory can be a future work.

Part II

Verification of Protocols

6

Mechanical Verification of a Two-Way Sliding Window Protocol

6.1 Introduction

A sliding window protocol [CK74] (SWP) ensures successful transmission of messages from a sender to a receiver through a medium, in which messages may get lost. Its main characteristic is that the sender does not wait for an incoming acknowledgement before sending next messages, for optimal use of bandwidth. Many data communication systems include a SWP, in one of its many variations.

In SWPs, both the sender and the receiver maintain a buffer. In practice the buffer at the receiver side is often much smaller than at the sender side, but here we make the simplifying assumption that both buffers can contain up to n messages. By providing the messages with sequence numbers, reliable in-order delivery without duplications is guaranteed. The sequence numbers can be taken modulo $2n$ (and not less, see [Tan81] for a nice argument). The messages at the sender are numbered from i to $i+n$ (modulo $2n$); this is called a *window*. When an acknowledgement reaches the sender, indicating that k messages have arrived correctly, the window *slides* forward, so that the sending buffer can contain messages with sequence numbers $i+k$ to $i+k+n$ (modulo $2n$). The window of the receiver slides forward when the first element in this window is passed on to the environment.

We consider a *two-way* SWP, in which both parties can both send and receive data elements from each other. One way of achieving full-duplex data transmission is to have two separate communication channels and use each one for simplex data traffic (in different directions). Then there are two separate physical circuits, each with a forward channel (for data) and a reverse channel (for acknowledgements). In both cases the bandwidth of the reverse channel is almost entirely wasted. In effect, the

user is paying for two circuits but using the capacity of one. A better idea is to use the same circuit in both directions. Each party maintains two buffers, for storing the two opposite data streams. In this two-way version of the SWP, an acknowledgement that is sent from one party to the other may get a free ride by attaching it to a data element. This method for efficiently passing acknowledgements and data elements through a channel in the same direction, which is known as *piggybacking*, is used broadly in transmission control protocols, see [Tan81]. The main advantage of piggybacking is a better use of available bandwidth. The extra acknowledgement field in the data frame costs only a few bits, whereas a separate acknowledgement would need a header and a checksum. In addition, fewer frames sent means fewer 'frame arrived' interrupts.

The main motivations for the current research are: (1) to provide a mechanised correctness proof of the most complicated version of the SWP in [Tan81], including the piggybacking mechanism; and (2) to gain experience in extending an existing PVS formalisation, namely the one from [FGP+04].

The structure of the proof is as follows. First, we linearize the specification, meaning that we get rid of parallel operators. Moreover, communication actions are stripped from their data parameters. Then we eliminate modulo arithmetic, using an idea from Schoone [Sch91]. Finally, we apply the cones and foci technique, to prove that the linear specification without modulo arithmetic is branching bisimilar to a pair of FIFO queues of capacity $2n$ and $2n_2$. The lemmas for the data types, the invariants, the transformations and the matching criteria have all been checked using PVS. The PVS files are available via `http://homepages.cwi.nl/~vdpol/piggybacking.html`.

The remainder of this chapter is set up as follows. Section 6.2 introduces the process part of μCRL. In Section 6.3, the data types needed for specifying the SWP and its external behaviour are presented. Section 6.4 features the μCRL specifications of the two-way SWP with piggybacking, and its external behaviour. In Section 6.5, three consecutive transformations are applied to the specification of the SWP, to linearize the specification, eliminate arguments of communication actions, and get rid of modulo arithmetic. In Section 6.6, properties of the data types and invariants of the transformed specification are formulated; their proofs are in the appendix. In Section 6.7, it is proved that the three transformations preserve branching bisimilarity, and that the transformed specification behaves as a pair of FIFO queues.

6.2 μCRL

μCRL [GP95] (see also [GR01]) is a language for specifying distributed systems and protocols in an algebraic style. It is based on the process algebra ACP [BK84] extended with equational abstract data types [LEW96]. We will use $\approx$ for equality between process terms and $=$ for equality between data terms.

A μCRL specification of data types consists of two parts: A signature of function symbols from which one can build data terms, and axioms that induce an equality

relation on data terms of the same type. They provide a loose semantics, meaning that it is allowed to have multiple models. The data types needed for our μCRL specification of a SWP are presented in Section 6.3. In particular we have the data sort of booleans *Bool* with constants `true` and `false`, and the usual connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$ and $\leftrightarrow$. For a boolean b, we abbreviate $b = \mathtt{true}$ to b and $b = \mathtt{false}$ to $\neg b$.

The process part of μCRL is specified using a number of pre-defined process algebraic operators, which we will present below. From these operators one can build process terms, which describe the order in which the atomic actions from a set $\mathcal{A}$ may happen. A process term consists of actions and recursion variables combined by the process algebraic operators. Actions and recursion variables may carry data parameters. There are two predefined actions outside $\mathcal{A}$: δ represents deadlock, and τ a hidden action. These two actions never carry data parameters.

Two elementary operators to construct processes are *sequential composition*, written $p{\cdot}q$, and *alternative composition*, written $p + q$. The process $p{\cdot}q$ first executes p, until p terminates, and then continues with executing q. The process $p + q$ non-deterministically behaves as either p or q. *Summation* $\sum_{d:D} p(d)$ provides the possibly infinite non-deterministic choice over a data type D. For example, $\sum_{n:Nat} a(n)$ can perform the action $a(n)$ for all natural numbers n. The *conditional* construct $p \lhd b \rhd q$, with b a data term of sort *Bool*, behaves as p if b and as q if $\neg b$. *Parallel composition* $p \parallel q$ performs the processes p and q in parallel; in other words, it consists of the arbitrary interleaving of actions of the processes p and q. For example, if there is no communication possible between actions a and b, then $a \parallel b$ behaves as $(a{\cdot}b) + (b{\cdot}a)$. Moreover, actions from p and q may also synchronise to a communication action, when this is explicitly allowed by a predefined *communication function*; two actions can only synchronise if their data parameters are equal. *Encapsulation* $\partial_{\mathcal{H}}(p)$, which renames all occurrences in p of actions from the set $\mathcal{H}$ into δ, can be used to force actions into communication. For example, if actions a and b communicate to c, then $\partial_{\{a,b\}}(a \parallel b) \approx c$. *Hiding* $\tau_{\mathcal{I}}(p)$ renames all occurrences in p of actions from the set $\mathcal{I}$ into τ. Finally, processes can be specified by means of recursive equations

$$X(d_1{:}D_1, \ldots, d_n{:}D_n) \approx p$$

where X is a recursion variable, d_i a data parameter of type D_i for $i = 1, \ldots, n$, and p a process term (possibly containing recursion variables and the parameters d_i). For example, let $X(n{:}Nat) \approx a(n){\cdot}X(n+1)$; then $X(0)$ can execute the infinite sequence of actions $a(0){\cdot}a(1){\cdot}a(2) \cdot \cdots$.

Definition 6.1 (Linear process equation) *A recursive specification is a linear process equation (LPE) if it is of the form*

$$X(d{:}D) \approx \sum_{j \in J} \sum_{e_j : E_j} a_j(f_j(d, e_j)){\cdot}X(g_j(d, e_j)) \lhd h_j(d, e_j) \rhd \delta$$

with J a finite index set, $f_j : D \times E_j \to D_j$, $g_j : D \times E_j \to D$, and $h_j : D \times E_j \to Bool$.

Note that an LPE does not contain parallel composition, encapsulation and hiding, and uses only one recursion variable. Groote, Ponse and Usenko [GPU01] presented a linearization algorithm that transforms μCRL specifications into LPEs.

The μCRL specification of the data part of a two-way SWP is presented in Section 6.3, while the process part is presented in Section 6.4.1. The μCRL specification of the external behaviour of this SWP, being a pair of FIFO queues, is presented in Section 6.4.2. Section 6.5.1 contains the LPE that results from applying the linearization algorithm to the μCRL specification of the SWP in Section 6.4.1

To each μCRL specification belongs a directed graph, called a labeled transition system. In this labeled transition system, the states are process terms, and the edges are labeled with parameterised actions. For example, given the μCRL specification $X(n{:}Nat) \approx a(n){\cdot}X(n+1)$, we have transitions $X(n) \stackrel{a(n)}{\to} X(n+1)$. Branching bisimilarity $\underline{\leftrightarrow}_b$ [vGW96] and strong bisimilarity $\underline{\leftrightarrow}$ [Par81] are two well-established equivalence relations on states in labeled transition systems.[1] Conveniently, strong bisimilarity implies branching bisimilarity. The proof theory of μCRL from [GP94] is sound with respect to branching bisimilarity, meaning that if $p \approx q$ can be derived from it then $p \underline{\leftrightarrow}_b q$.

Definition 6.2 (Branching bisimulation) *Given a labeled transition system. A* strong bisimulation relation *$\mathcal{B}$ is a symmetric binary relation on states such that if $s\,\mathcal{B}\,t$ and $s \stackrel{\ell}{\to} s'$, then there exists t' such that $t \stackrel{\ell}{\to} t'$ and $s'\,\mathcal{B}\,t'$. Two states s and t are* strongly bisimilar*, denoted by $s \underline{\leftrightarrow} t$, if there is a strong bisimulation relation $\mathcal{B}$ such that $s\,\mathcal{B}\,t$.*

A branching bisimulation relation *$\mathcal{B}$ is a symmetric binary relation on states such that if $s\,\mathcal{B}\,t$ and $s \stackrel{\ell}{\to} s'$, then*

- *either $\ell = \tau$ and $s'\,\mathcal{B}\,t$;*
- *or there is a sequence of (zero or more) τ-transitions $t \stackrel{\tau}{\to} \cdots \stackrel{\tau}{\to} \hat{t}$ such that $s\,\mathcal{B}\,\hat{t}$ and $\hat{t} \stackrel{\ell}{\to} t'$ with $s'\,\mathcal{B}\,t'$.*

Two states s and t are branching bisimilar*, denoted by $s \underline{\leftrightarrow}_b t$, if there is a branching bisimulation relation $\mathcal{B}$ such that $s\,\mathcal{B}\,t$.*

See [vG94] for a lucid exposition on why branching bisimilarity constitutes a sensible equivalence relation for concurrent processes.

The goal of this chapter is to prove that the initial state of the forthcoming μCRL specification of a two-way SWP is branching bisimilar to a pair of FIFO queues. In

[1] The definitions of these relations often take into account a special predicate on states to denote successful termination. This predicate is missing here, as successful termination does not play a role in our SWP specification.

the proof of this fact, in Section 6.7, we will use three proof techniques to derive that two μCRL specifications are branching (or even strongly) bisimilar: invariants, bisimulation criteria, and cones and foci.

An *invariant* $I : D \to Bool$ [BG94b] characterises the set of reachable states of an LPE $X(d{:}D)$. That is, if $I(d) = \texttt{true}$ and X can evolve from d to d' in zero or more transitions, then $I(d') = \texttt{true}$.

Definition 6.3 (Invariant) *$I : D \to Bool$ is an invariant for an LPE in Definition 6.1 if for all $d{:}D$, $j \in J$ and $e_j{:}E_j$,*

$$(I(d) \wedge h_j(d, e_j)) \;\to\; I(g_j(d, e_j)).$$

If I holds in a state d and $X(d)$ can perform a transition, meaning that $h_j(d, e_j) = \texttt{true}$ for some $e_j{:}E$, then it is ensured by the definition above that I holds in the resulting state $g_j(d, e_j)$.

Bisimulation criteria rephrase the question whether $X(d)$ and $Y(d')$ are strongly bisimilar in terms of data equalities, where $X(d{:}D)$ and $Y(d'{:}D')$ are LPEs. A *state mapping* ϕ relates each state in $X(d)$ to a state in $Y(d')$. If a number of bisimulation criteria are satisfied, then ϕ establishes a strong bisimulation relation between terms $X(d)$ and $Y(\phi(d))$.

Definition 6.4 (Bisimulation criteria) *Given two LPEs,*

$$X(d{:}D) \quad\approx\quad \textstyle\sum_{j\in J}\sum_{e_j:E_j} a_j(f_j(d, e_j)){\cdot}X(g_j(d, e_j)) \lhd h_j(d, e_j) \rhd \delta$$

$$Y(d'{:}D') \quad\approx\quad \textstyle\sum_{j\in J}\sum_{e'_j:E'_j} a_j(f'_j(d', e'_j)){\cdot}X(g'_j(d', e'_j)) \lhd h'_j(d', e'_j) \rhd \delta$$

and an invariant $I : D \to Bool$ for X. A state mapping *$\phi : D \to D'$ and* local mappings *$\psi_j : E_j \to E'_j$ for $j \in J$ satisfy the* bisimulation criteria *if for all states $d \in D$ in which invariant I holds:*

I $\quad \forall j{\in}J\,\forall e_j{:}E_j\ (h_j(d, e_j) \leftrightarrow h'_j(\phi(d), \psi_j(e_j))),$

II $\quad \forall j{\in}J\,\forall e_j{:}E_j\ (h_j(d, e_j) \wedge I(d)) \to (a_j(f_j(d, e_j)) = a_j(f'_j(\phi(d), \psi_j(e_j)))),$

III $\quad \forall j{\in}J\,\forall e_j{:}E_j\ (h_j(d, e_j) \wedge I(d)) \to (\phi(g_j(d, e_j)) = g'_j(\phi(d), \psi_j(e_j))).$

Criterion I expresses that at each summand i, the corresponding guard of X holds if and only if the corresponding guard of Y holds with parameters $(\phi(d), \psi_j(e_j))$. Criterion II (III) states that at any summand i, the corresponding action (next state, after applying ϕ on it) of X could be equated to the corresponding action (next state) of Y with parameters $(\phi(d), \psi_j(e_j))$.

Theorem 6.5 (Bisimulation criteria) *Given two LPEs $X(d{:}D)$ and $Y(d'{:}D')$ written as in Definition 6.4, and $I : D \to Bool$ an invariant for X. Let $\phi : D \to D'$ and $\psi_j : E_j \to E'_j$ for $j \in J$ satisfy the bisimulation criteria in Definition 6.4. Then $X(d) \underline{\leftrightarrow} Y(\phi(d))$ for all $d \in D$ in which I holds.*

This theorem has been proved in PVS. The proof is available at `http://homepages.cwi.nl/~vdpol/piggybacking.html`.

The *cones and foci* method from [GS01, FP03] rephrases the question whether $\tau_{\mathcal{I}}(X(d))$ and $Y(d')$ are branching bisimilar in terms of data equalities, where $X(d{:}D)$ and $Y(d'{:}D')$ are LPEs, and the latter LPE does not contain actions from some set $\mathcal{I}$ of internal actions. A *state mapping* ϕ relates each state in $X(d)$ to a state in $Y(d')$. Furthermore, some $d{:}D$ are declared to be *focus points*. The *cone* of a focus point consists of the states in $X(d)$ that can reach this focus point by a string of actions from $\mathcal{I}$. It is required that each reachable state in $X(d)$ is in the cone of a focus point. If a number of *matching criteria* are satisfied, then ϕ establishes a branching bisimulation relation between terms $\tau_{\mathcal{I}}(X(d))$ and $Y(\phi(d))$.

For example, consider the LPEs $X(b{:}Bool) \approx a{\cdot}X(\neg b) \lhd b \rhd \delta + c{\cdot}X(b) \lhd \neg b \rhd \delta$ and $Y(d'{:}D') \approx a{\cdot}Y(d')$, with $\mathcal{I} = \{c\}$ and focus point $\mathtt{true}$. Moreover, $X(\mathtt{false}) \xrightarrow{c} X(\mathtt{true})$, i.e., $\mathtt{false}$ can reach the focus point in a single c-transition. For any $d'{:}D'$, the state mapping $\phi(b) = d'$ for $b{:}Bool$ satisfies the matching criteria.

Given an invariant I, only $d{:}D$ with $I(d) = \mathtt{true}$ need to be in the cone of a focus point, and we only need to satisfy the matching criteria for $d{:}D$ with $I(d) = \mathtt{true}$.

Definition 6.6 (Matching criteria) *Given two LPEs:*

$$X(d{:}D) \approx \sum_{j\in J} \sum_{e_j:E_j} a_j(f_j(d,e_j)){\cdot}X(g_j(d,e_j)) \lhd h_j(d,e_j) \rhd \delta$$

$$Y(d'{:}D') \approx \sum_{\{j\in J|a_j\notin\mathcal{I}\}} \sum_{e_j:E_j} a_j(f'_j(d',e_j)){\cdot}Y(g'_j(d',e_j)) \lhd h'_j(d',e_j) \rhd \delta$$

Let FC: $D \to Bool$ be a prediacate which designates the focus points. A state mapping $\phi : D \to D'$ satisfies the matching criteria *for $d{:}D$ if for all $j \in J$ with $a_j \notin \mathcal{I}$ and all $k \in J$ with $a_k \in \mathcal{I}$:*

I $\quad \forall e_k{:}E_k\, (h_k(d,e_k) \to \phi(d) = \phi(g_k(d,e_k)))$;

II $\quad \forall e_j{:}E_j\, (h_j(d,e_j) \to h'_j(\phi(d),e_j))$;

III $\quad FC(d) \to \forall e_j{:}E_j\, (h'_j(\phi(d),e_j) \to h_j(d,e_j))$;

IV $\quad \forall e_j{:}E_j\, (h_j(d,e_j) \to f_j(d,e_j) = f'_j(\phi(d),e_j))$;

V $\quad \forall e_j{:}E_j\, (h_j(d,e_j) \to \phi(g_j(d,e_j)) = g'_j(\phi(d),e_j))$.

Matching criterion I requires that the internal transitions at d are inert, meaning that d and $g_k(d, e_k)$ are branching bisimilar. Criteria II, IV and V express that each external transition of d can be simulated by $\phi(d)$. Finally, criterion III expresses that if d is a focus point, then each external transition of $\phi(d)$ can be simulated by d.

Theorem 6.7 (Cones and foci) *Given LPEs $X(d{:}D)$ and $Y(d'{:}D')$ written as in Definition 6.6. Let $I : D \to Bool$ be an invariant for X. Suppose that for all $d{:}D$ with $I(d)$:*

1. *$\phi : D \to D'$ satisfies the matching criteria for d; and*
2. *there is a $\hat{d}{:}D$ such that $FC(\hat{d})$ and X can perform transitions $d \stackrel{c_1}{\to} \cdots \stackrel{c_k}{\to} \hat{d}$ with $c_1, \ldots, c_k \in \mathcal{I}$.*

Then for all $d{:}D$ with $I(d)$, $\tau_{\mathcal{I}}(X(d)) \leftrightarrow_b Y(\phi(d))$.

This theorem has been proved in PVS, see [FPvdP05].

6.3 Data Types

In this section, the data types used in the μCRL specification of the two-way SWP are presented: booleans, natural numbers supplied with modulo arithmetic, buffers, and lists. Furthermore, basic properties are given for the operations defined on these data types.

Most of this data specification was taken directly from [BFG$^+$05], which means that in Section 6.6 we will be able to reuse large parts of the data lemmas and proofs from [BFG$^+$05]. Only the definition of the *next-empty*$|_n$ function differs from the one in [BFG$^+$05]; our definition always produces a result, while the one in [BFG$^+$05] does not, which is a problem for the formalisation in PVS. Moreover, the functions *add*, $\|_n$, *smaller* and *sort* about sorted buffers do not occur in [BFG$^+$05].

6.3.1 Booleans

We introduce the data type *Bool* of booleans.

`sort` :	$Bool$
`cons` :	$\mathtt{true},\ \mathtt{false} :\to Bool$
`func` :	$\neg : Bool \to Bool$
	$\vee,\ \wedge : Bool \times Bool \to Bool$
	$\to,\ \leftrightarrow : Bool \times Bool \to Bool$
`var` :	$b,\ c : Bool$
`rew` :	$\neg\mathtt{true} = \mathtt{false}$
	$\neg\mathtt{false} = \mathtt{true}$
	$b \wedge \mathtt{true} = b$

$$b \wedge \texttt{false} = \texttt{false}$$
$$b \vee \texttt{true} = \texttt{true}$$
$$b \vee \texttt{false} = b$$
$$b \rightarrow c = c \vee \neg b$$
$$b \leftrightarrow c = (b \rightarrow c) \wedge (c \rightarrow b)$$

$\wedge$ and $\vee$ represent conjunction and disjunction, $\rightarrow$ and $\leftrightarrow$ denote implication and bi-implication, and $\neg$ denotes negation.

Furthermore for every given sort D we assume a function *if* which represents an If-Then-Else operation:

if :	$Bool \times D \times D \rightarrow D$
`var` :	$d,\ e$
`rew` :	$\mathit{if}(\texttt{true}, d, e) = d$
	$\mathit{if}(\texttt{false}, d, e) = e$

Finally, for each data type D in this chapter, one can easily define a mapping $eq : D \times D \rightarrow Bool$ such that $eq(d, e)$ holds if and only if $d = e$ can be derived. For notational convenience we take the liberty to write $d = e$ instead of $eq(d, e)$.

6.3.2 Natural Numbers

Below we specify the data type natural numbers.

`sort` :	Nat
`cons` :	$0 :\rightarrow Nat$
`func` :	$S : Nat \rightarrow Nat$
	$+, \dot{-}, \cdot : Nat \times Nat \rightarrow Nat$
	$\leq, <, \geq, >: Nat \times Nat \rightarrow Bool$
	$\mid : Nat \times Nat \rightarrow Nat$
	$div : Nat \times Nat \rightarrow Nat$
`var` :	$i,\ j,\ n : Nat$
`rew` :	$i + 0 = i$
	$i + S(j) = S(i + j)$
	$i \dot{-} 0 = i$
	$0 \dot{-} i = 0$
	$S(i) \dot{-} S(j) = i \dot{-} j$
	$i \cdot 0 = 0$
	$i \cdot S(j) = (i \cdot j) + i$
	$0 \leq i = \texttt{true}$
	$S(i) \leq 0 = \texttt{false}$
	$S(i) \leq S(j) = i \leq j$
	$0 < S(i) = \texttt{true}$

$$
\begin{aligned}
&i < 0 = \texttt{false}\\
&S(i) < S(j) = i < j\\
&i \geq j = \neg(j < i)\\
&i > j = \neg(j \leq i)\\
&i|_n = \mathit{if}(i < n, i, (i \dot{-} n)|_n)\\
&i\ \mathit{div}\ n = \mathit{if}(i < n, 0, S((i \dot{-} n)\ \mathit{div}\ n))
\end{aligned}
$$

0 denotes zero and $S(n)$ the successor of n. The infix operations $+$, $\dot{-}$ and $\cdot$ represent addition, monus (also called cut-off subtraction) and multiplication, respectively. The infix operations $\leq$, $<$, $\geq$ and $>$ are the less-than(-or-equal) and greater-than(-or-equal) operations.

Since the buffers at the sender and the receiver in the SWP are of finite size, modulo calculations will play an important role. $i|_n$ denotes i modulo n, while $i\ \mathit{div}\ n$ denotes i integer divided by n.

In the proofs we will take notational liberties like omitting the sign for multiplication, and abbreviating $\neg(i = j)$ to $i \neq j$, $(k < \ell) \wedge (\ell < m)$ to $k < \ell < m$, $S(0)$ to 1, and $S(S(0))$ to 2.

We will use induction schemes to prove some properties about data types. Below we formulate two of them.

Definition 6.8 *(Standard induction) For any $f : \mathit{Nat} \to \mathit{Bool}$,*

$$(f(0)\ \wedge\ \forall m{:}\mathit{Nat}\,(f(m) \to f(S(m))))\ \to\ \forall n{:}\mathit{Nat}\, f(n)$$

Definition 6.9 *For any $f : \mathit{Nat} \to \mathit{Bool}$,*

$$(f(0)\ \wedge\ \forall m{:}\mathit{Nat}\ f(S(m)))\ \to\ \forall n{:}\mathit{Nat}\ f(n)$$

6.3.3 Buffers

The two parties in the two-way SWP will both maintain two buffers containing the sending and the receiving window (outside these windows both buffers will be empty).

$$
\begin{aligned}
\texttt{cons}:\ & [] :\to \mathit{Buf}\\
\texttt{func}:\ & \mathit{inb} : \Delta \times \mathit{Nat} \times \mathit{Buf} \to \mathit{Buf}\\
& \mathit{add} : \Delta \times \mathit{Nat} \times \mathit{Buf} \to \mathit{Buf}\\
& | : \mathit{Buf} \times \mathit{Nat} \to \mathit{Buf}\\
& \| : \mathit{Buf} \times \mathit{Nat} \to \mathit{Buf}\\
& \mathit{smaller} : \mathit{Nat} \times \mathit{Buf} \to \mathit{Bool}\\
& \mathit{sort} : \mathit{Buf} \to \mathit{Bool}\\
& \mathit{test} : \mathit{Nat} \times \mathit{Buf} \to \mathit{Bool}\\
& \mathit{retrieve} : \mathit{Nat} \times \mathit{Buf} \to \Delta\\
& \mathit{remove} : \mathit{Nat} \times \mathit{Buf} \to \mathit{Buf}
\end{aligned}
$$

$release, release|_n : Nat \times Nat \times Buf \to Buf$
$next\text{-}empty, next\text{-}empty|_n : Nat \times Buf \to Nat$
$in\text{-}window : Nat \times Nat \times Nat \to Bool$
$max : Buf \to Nat$

`var` : $i,\ j,\ n : Nat$
$q : Buf$
$d,\ e : \Delta$

`rew` : $add(d, i, []) = inb(d, i, [])$
$add(d, i, inb(e, j, q)) = if(i{>}j, inb(e, j, add(d, i, q)), inb(d, i, remove(i, inb(e, j, q))))$
$[]|_n = []$
$inb(d, i, q)|_n = inb(d, i|_n, q|_n)$
$[]\|_n = []$
$inb(d, i, q)\|_n = add(d, i|_n, q\|_n)$
$smaller(i, []) = \texttt{true}$
$smaller(i, inb(d, j, q)) = i < j \wedge smaller(i, q)$
$sort([]) = \texttt{true}$
$sort(inb(d, j, q)) = smaller(j, q) \wedge sort(q)$
$test(i, []) = \texttt{false}$
$test(i, inb(d, j, q)) = i{=}j \vee test(i, q)$
$retrieve(i, inb(d, j, q)) = if(i{=}j, d, retrieve(i, q))$
$remove(i, []) = []$
$remove(i, inb(d, j, q)) = if(i{=}j, remove(i, q), inb(d, j, remove(i, q)))$
$release(i, j, q) = if(i \geq j, q, release(S(i), j, remove(i, q)))$
$release|_n(i, j, q) = if(i|_n{=}j|_n, q, release|_n(S(i), j, remove(i|_n, q)))$
$next\text{-}empty(i, q) = if(test(i, q), next\text{-}empty(S(i), q), i)$
$next\text{-}empty|_n(i, q) = if(next\text{-}empty(i|_n, q) < n, next\text{-}empty(i|_n, q), if(next\text{-}empty(0, q) < n, next\text{-}empty(0, q), n))$
$in\text{-}window(i, j, k) = i \leq j < k \vee k < i \leq j \vee j < k < i$
$max([]) = 0$
$max(inb(d, i, q)) = if(i \geq max(q), i, max(q))$

Δ represents the set of data elements that can be communicated between the two parties. The buffers are modeled as a list of pairs (d, i) with $d{:}\Delta$ and $i{:}Nat$, representing that cell (or sequence number) i of the buffer is occupied by datum d; cells for which no datum is specified are empty. The empty buffer is denoted by [], and $inb(d, i, q)$ is the buffer that is obtained from q by placing datum d in cell i, possibly overwriting the previous datum in cell i (if this cell was not empty).

add is similar to *inb*, except that given a sorted buffer q without duplications, $add(d, i, q)$ is again sorted without duplications. In $q|_n$, the sequence numbers in q are taken modulo n, and in $q\|_n$ the resulting buffer is moreover sorted without

duplications. *sort* checks whether a buffer is sorted, and *smaller* is a help function that is needed in the definition of *sort*.

$test(i, q)$ produces `true` if and only if cell i in q is occupied, $retrieve(i, q)$ produces the datum that resides at cell i in buffer q (if this cell is occupied),[2] and $remove(i, q)$ is obtained by emptying cell i in buffer q. $release(i, j, q)$ is obtained by emptying cells i up to but not including j in q, and $release|_n(i, j, q)$ does the same modulo n. $next\text{-}empty(i, q)$ produces the first empty cell in q, counting upwards from sequence number i onward, and $next\text{-}empty|_n(i, q)$ does the same modulo n. $in\text{-}window(i, j, k)$ produces `true` if and only if j lies in the range from i to $k \dot{-} 1$, modulo n, for n greater than i, j and k. Finally, $max(q)$ produces the greatest sequence number that is occupied in q.

6.3.4 Lists

We introduce the data type of *List* of lists, which will be used in the specification of the desired external behaviour of the SWP: a pair of FIFO queues of size $2n$.

$$
\begin{array}{ll}
\texttt{cons}: & \langle\rangle :\to List \\
 & d_0 : \Delta \\
\texttt{func}: & inl : \Delta \times List \to List \\
 & length : List \to Nat \\
 & top : List \to \Delta \\
 & tail : List \to List \\
 & append : \Delta \times List \to List \\
 & ++ : List \times List \to List \\
\texttt{var}: & \lambda,\ \lambda' : List \\
 & d,\ e : \Delta \\
\texttt{rew}: & length(\langle\rangle) = 0 \\
 & length(inl(d, \lambda)) = S(length(\lambda)) \\
 & top(inl(d, \lambda)) = d \\
 & tail(inl(d, \lambda)) = \lambda \\
 & append(d, \langle\rangle) = inl(d, \langle\rangle) \\
 & append(d, inl(e, \lambda)) = inl(e, append(d, \lambda)) \\
 & \langle\rangle ++ \lambda = \lambda \\
 & inl(d, \lambda) ++ \lambda' = inl(d, \lambda ++ \lambda') \\
 & q[i..j\rangle = if(i \geq j, \langle\rangle, inl(retrieve(i, q), q[S(i)..j\rangle))
\end{array}
$$

[2] Note that $retrieve(i, [])$ is undefined. One could choose to equate it to a default value in Δ, or to a fresh error element in Δ. However, with the first approach an occurrence of $retrieve(i, [])$ might remain undetected, and the second approach would needlessly complicate the data type Δ. We prefer to work with an underspecified version of *retrieve*, which is allowed in μCRL, since data types have a loose semantics (i.e. it is allowed to have partial functions). All operations in μCRL data models, however, are total; underspecified operations lead to the existence of multiple models.

$\langle\rangle$ denotes the empty list, and $inl(d, \lambda)$ adds datum d at the top of list λ. A special datum d_0 is specified to serve as a dummy value for data parameters. $length(\lambda)$ denotes the length of λ, $top(\lambda)$ produces the datum that resides at the top of λ, $tail(\lambda)$ is obtained by removing the top position in λ, $append(d, \lambda)$ adds datum d at the end of λ, and $\lambda{+\!\!+}\lambda'$ represents list concatenation. Finally, $q[i..j\rangle$ is the list containing the elements in buffer q at positions i up to but not including j.

6.4 A Two-Way Sliding Window Protocol with Piggybacking

This section contains a μCRL specification of a two-way SWP with piggybacking, together with its desired external behaviour. Figure 6.1 depicts the two-way SWP. First, we explain only one direction of it, being the one-way version of this SWP, without piggybacking, from [FGP+04].

A sender, denoted by **S**/**R**, stores data elements that it receives via channel A in a buffer of size $2n$, in the order in which they are received. It can send a datum, together with its sequence number in the buffer, to a receiver **R**/**S** via a medium that behaves as lossy queue of capacity one, represented by the medium **K** and the channels B and C. Upon reception, the receiver may store the datum in its buffer, where its position in the buffer is dictated by the attached sequence number. In order to avoid a possible overlap between the sequence numbers of different data elements in the buffers of sender and receiver, no more than one half of each of these two buffers may be occupied at any time; these halves are called the sending and the receiving window, respectively. The receiver can pass on a datum that resides at the first cell in its window via channel D; in that case the receiving window slides forward by one cell. Furthermore, the receiver can send the sequence number of the first empty cell in (or just outside) its window as an acknowledgement to the sender via a medium that behaves as lossy queue of capacity one, represented by the medium **L** and the channels E and F. If the sender receives this acknowledgement, its window slides forward accordingly.

In the two-way SWP, there are data streams in both directions, so **S**/**R** and **R**/**S** are both playing the roles of sender and receiver. Furthermore, when a datum arrives, instead of immediately sending an acknowledgement, the receiver may restrain himself and wait until the network layer passes on the next datum. The acknowledgement is then attached to the outgoing datum, so that the acknowledgement gets a free ride. This is known as *piggybacking*.

6.4.1 Specification of the Sliding Window Protocol

The sender/receiver **S**/**R** is modeled by the process $\mathbf{S}/\mathbf{R}(\ell, m, n, n_2, q, q_2', \ell_2')$, where q is its sending buffer of size $2n$, ℓ is the first cell in the window of q, and m the first empty cell in (or just outside) this window. Furthermore, q_2' is the receiving buffer of

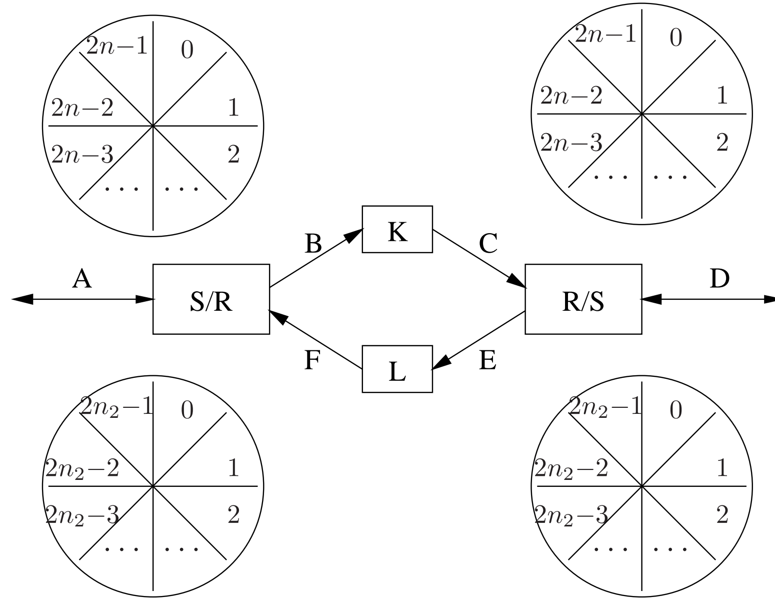

Figure 6.1: Sliding window protocol

size $2n_2$, and ℓ'_2 is the first cell in the window of q_2.

The μCRL specification of **S**/**R** consists of seven clauses. The first clause of the specification expresses that **S**/**R** can receive a datum via channel A and place it in its sending window, under the condition that this window is not yet full. The next two clauses specify that **S**/**R** can receive a datum/acknowledgement pair via channel F; the data part is either added to q_2 if it is within the receiving window (second clause), or ignored if it is outside this window (third clause). In both clauses, q is emptied from ℓ up to but not including the received acknowledgement. The fourth clause specifies the reception of a single (i.e., non-piggybacked) acknowledgement. According to the fifth clause, data elements for transmission via channel B are taken (at random) from the filled part of the sending window; the first empty position in (or just outside) the receiving window is attached to this datum as an acknowledgement. In the sixth clause, **S**/**R** sends a single acknowledgement. Finally, clause seven expresses that if the first cell in the receiving window is occupied, then **S**/**R** can send this datum into

channel A, after which the cell is emptied.

$$
\begin{array}{ll}
& \mathbf{S}/\mathbf{R}(\ell{:}Nat, m{:}Nat, n{:}Nat, n_2{:}Nat, q{:}Buf, q_2'{:}Buf, \ell_2'{:}Nat) \\
\approx & \sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot\mathbf{S}/\mathbf{R}(\ell, S(m)|_{2n}, n, n_2, add(d,m,q), q_2', \ell_2') \\
& \qquad \lhd\ in\text{-}window(\ell, m, (\ell+n)|_{2n})\ \rhd \delta \\
+ & \sum_{d:\Delta}\sum_{i:Nat}\sum_{k:Nat} r_{\mathrm{F}}(d,i,k)\cdot\mathbf{S}/\mathbf{R}(k, m, n, n_2, release|_{2n}(\ell,k,q), add(d,i,q_2'), \ell_2') \\
& \qquad \lhd\ in\text{-}window(\ell_2', i, (\ell_2'+n_2)|_{2n_2})\ \rhd \delta \\
+ & \sum_{d:\Delta}\sum_{i:Nat}\sum_{k:Nat} r_{\mathrm{F}}(d,i,k)\cdot\mathbf{S}/\mathbf{R}(k, m, n, n_2, release|_{2n}(\ell,k,q), q_2', \ell_2') \\
& \qquad \lhd\ \neg in\text{-}window(\ell_2', i, (\ell_2'+n_2)|_{2n_2})\ \rhd \delta \\
+ & \sum_{k:Nat} r_{\mathrm{F}}(k)\cdot\mathbf{S}/\mathbf{R}(k, m, n, n_2, release|_{2n}(\ell,k,q), q_2', \ell_2') \\
+ & \sum_{k:Nat} s_{\mathrm{B}}(retrieve(k,q), k, next\text{-}empty|_{2n_2}(\ell_2', q_2'))\cdot\mathbf{S}/\mathbf{R}(\ell, m, n, n_2, q, q_2', \ell_2') \\
& \qquad \lhd\ test(k,q)\ \rhd \delta \\
+ & s_{\mathrm{B}}(next\text{-}empty|_{2n_2}(\ell_2', q_2'))\cdot\mathbf{S}/\mathbf{R}(\ell, m, n, n_2, q, q_2', \ell_2') \\
+ & s_{\mathrm{A}}(retrieve(\ell_2', q_2'))\cdot\mathbf{S}/\mathbf{R}(\ell, m, n, n_2, q, remove(\ell_2', q_2'), S(\ell_2')|_{2n_2}) \\
& \qquad \lhd\ test(\ell_2', q_2')\ \rhd \delta
\end{array}
$$

The μCRL specification of $\mathbf{R}/\mathbf{S}$ is symmetrical to the one of $\mathbf{S}/\mathbf{R}$. In the process $\mathbf{R}/\mathbf{S}(\ell_2, m_2, n, n_2, q_2, q', \ell')$, q' is the receiving buffer of size $2n$, and ℓ' is the first position in the window of q. Furthermore, q_2 is the sending buffer of size $2n_2$, ℓ_2 is the first position in the window of q_2, and m_2 the first empty position in (or just outside) this window.

$$
\begin{array}{ll}
& \mathbf{R}/\mathbf{S}(\ell_2{:}Nat, m_2{:}Nat, n{:}Nat, n_2{:}Nat, q_2{:}Buf, q'{:}Buf, \ell'{:}Nat) \\
\approx & \sum_{d:\Delta} r_{\mathrm{D}}(d)\cdot\mathbf{R}/\mathbf{S}(\ell_2, S(m_2)|_{2n_2}, n, n_2, add(d,m_2,q_2), q', \ell') \\
& \qquad \lhd\ in\text{-}window(\ell_2, m_2, (\ell_2+n_2)|_{2n_2})\ \rhd \delta \\
+ & \sum_{d:\Delta}\sum_{i:Nat}\sum_{k:Nat} r_{\mathrm{C}}(d,i,k)\cdot\mathbf{R}/\mathbf{S}(k, m_2, n, n_2, release|_{2n_2}(\ell_2,k,q_2), add(d,i,q'), \ell') \\
& \qquad \lhd\ in\text{-}window(\ell', i, (\ell'+n)|_{2n})\ \rhd \delta \\
+ & \sum_{d:\Delta}\sum_{i:Nat}\sum_{k:Nat} r_{\mathrm{C}}(d,i,k)\cdot\mathbf{R}/\mathbf{S}(k, m_2, n, n_2, release|_{2n_2}(\ell_2,k,q_2), q', \ell') \\
& \qquad \lhd\ \neg in\text{-}window(\ell', i, (\ell'+n)|_{2n})\ \rhd \delta \\
+ & \sum_{k:Nat} r_{\mathrm{C}}(k)\cdot\mathbf{R}/\mathbf{S}(k, m_2, n, n_2, release|_{2n_2}(\ell_2,k,q_2), q', \ell') \\
+ & \sum_{k:Nat} s_{\mathrm{E}}(retrieve(k,q_2), k, next\text{-}empty|_{2n}(\ell', q'))\cdot\mathbf{R}/\mathbf{S}(\ell_2, m_2, n, n_2, q_2, q', \ell') \\
& \qquad \lhd\ test(k,q_2)\ \rhd \delta \\
+ & s_{\mathrm{E}}(next\text{-}empty|_{2n}(\ell', q'))\cdot\mathbf{R}/\mathbf{S}(\ell_2, m_2, n, n_2, q_2, q', \ell') \\
+ & s_{\mathrm{D}}(retrieve(\ell', q'))\cdot\mathbf{R}/\mathbf{S}(\ell_2, m_2, n, n_2, q_2, remove(\ell', q'), S(\ell')|_{2n}) \\
& \qquad \lhd\ test(\ell', q')\ \rhd \delta
\end{array}
$$

Finally, we specify the mediums **K** and **L**, which have capacity one and may lose frames and acknowledgements.

$$\begin{array}{rcl} \mathbf{K} & \approx & \sum_{d:\Delta}\sum_{k:Nat}\sum_{i:Nat} r_{\mathrm{B}}(d,k,i)\cdot(j\cdot s_{\mathrm{C}}(d,k,i)+j)\cdot\mathbf{K} \\ & + & \sum_{i:Nat} r_{\mathrm{B}}(i)\cdot(j\cdot s_{\mathrm{C}}(i)+j)\cdot\mathbf{K} \end{array}$$

$$\begin{array}{rcl} \mathbf{L} & \approx & \sum_{d:\Delta}\sum_{k:Nat}\sum_{i:Nat} r_{\mathrm{E}}(d,k,i)\cdot(j\cdot s_{\mathrm{F}}(d,k,i)+j)\cdot\mathbf{L} \\ & + & \sum_{i:Nat} r_{\mathrm{E}}(i)\cdot(j\cdot s_{\mathrm{F}}(i)+j)\cdot\mathbf{L} \end{array}$$

For each channel $i \in \{\mathrm{B},\mathrm{C},\mathrm{E},\mathrm{F}\}$, actions s_i and r_i can communicate, resulting in the action c_i. The initial state of the SWP is expressed by

$$\tau_{\mathcal{I}}(\partial_{\mathcal{H}}(\mathbf{S}/\mathbf{R}(0,0,n,n_2,[],[],0) \parallel \mathbf{R}/\mathbf{S}(0,0,n,n_2,[],[],0) \parallel \mathbf{K} \parallel \mathbf{L}))$$

where the set $\mathcal{H}$ consists of the read and send actions over the internal channels B, C, E, and F, namely $\mathcal{H} = \{s_{\mathrm{B}}, r_{\mathrm{B}}, s_{\mathrm{C}}, r_{\mathrm{C}}, s_{\mathrm{E}}, r_{\mathrm{E}}, s_{\mathrm{F}}, r_{\mathrm{F}}\}$, while the set $\mathcal{I}$ consists of the communication actions over these internal channels together with j, namely $\mathcal{I} = \{c_{\mathrm{B}}, c_{\mathrm{C}}, c_{\mathrm{E}}, c_{\mathrm{F}}, j\}$.

6.4.2 External Behaviour

Data elements that are read from channel A should be sent into channel D in the same order, and vice versa data elements that are read from channel D should be sent into channel A in the same order. No data elements should be lost. In other words, the SWP is intended to be a solution for the following linear μCRL specification, representing a pair of FIFO queues of capacity $2n$ and $2n_2$.

$$\begin{array}{rcl} \mathbf{Z}(\lambda_1{:}List, \lambda_2{:}List) & \approx & \sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot\mathbf{Z}(append(d,\lambda_1),\lambda_2) \lhd length(\lambda_1) < 2n \rhd \delta \\ & + & s_{\mathrm{D}}(top(\lambda_1))\cdot\mathbf{Z}(tail(\lambda_1),\lambda_2) \lhd length(\lambda_1) > 0 \rhd \delta \\ & + & \sum_{d:\Delta} r_{\mathrm{D}}(d)\cdot\mathbf{Z}(\lambda_1, append(d,\lambda_2)) \lhd length(\lambda_2) < 2n_2 \rhd \delta \\ & + & s_{\mathrm{A}}(top(\lambda_2))\cdot\mathbf{Z}(\lambda_1, tail(\lambda_2)) \lhd length(\lambda_2) > 0 \rhd \delta \end{array}$$

Note that $r_{\mathrm{A}}(d)$ can be performed until the list λ_1 contains $2n$ elements, because in that situation the sending window of **S/R** and the receiving window of **R/S** will be filled. Furthermore, $s_{\mathrm{D}}(top(\lambda_1))$ can only be performed if λ_1 is not empty. Likewise, $r_{\mathrm{D}}(d)$ can be performed until the list λ_2 contains $2n_2$ elements, and $s_{\mathrm{A}}(top(\lambda_2))$ can only be performed if λ_2 is not empty.

The remainder of this chapter is devoted to proving the following theorem, expressing that the external behaviour of our μCRL specification of a two-way SWP with piggybacking corresponds to a pair of FIFO queues of capacities $2n$ and $2n_2$.

Theorem 6.10 (Correctness of two-way SWP)
$\tau_{\mathcal{I}}(\partial_{\mathcal{H}}(\mathbf{S}/\mathbf{R}(0,0,n,n_2,[],[],0) \parallel \mathbf{R}/\mathbf{S}(0,0,n,n_2,[],[],0) \parallel \mathbf{K} \parallel \mathbf{L})) \;\underline{\leftrightarrow}_b\; \mathbf{Z}(\langle\rangle,\langle\rangle)$

6.5 Transformations of the Specification

This section witnesses three transformations, one to eliminate parallel operators, one to eliminate arguments of communication actions, and one to eliminate modulo arithmetic.

6.5.1 Linearisation

The starting point of our correctness proof is a linear specification $\mathbf{M}_{mod}$, in which no parallel composition, encapsulation and hiding operators occur. $\mathbf{M}_{mod}$ can be obtained from the μCRL specification of the SWP without the hiding operator, i.e.,

$$\partial_{\mathcal{H}}(\mathbf{S}/\mathbf{R}(0,0,n,n_2,[],[],0) \parallel \mathbf{R}/\mathbf{S}(0,0,n,n_2,[],[],0) \parallel \mathbf{K} \parallel \mathbf{L})$$

by means of the linearization algorithm presented in [GPU01]. The specification $\mathbf{M}_{mod}$ was generated automatically using the μCRL toolset.

$\mathbf{M}_{mod}$ contains eight extra parameters: $e, e_2{:}D$ and $g, g', h, h', h_2, h_2'{:}Nat$. Intuitively, g is 5 when medium $\mathbf{K}$ is inactive, is 4 or 2 when $\mathbf{K}$ just received a data frame or a single acknowledgement, respectively, and is 3 or 1 when $\mathbf{K}$ has decided to pass on this data frame or acknowledgement, respectively. The parameters e, h and h_2' represent the memory of $\mathbf{K}$, meaning that they can store the datum that is being sent from $\mathbf{S}/\mathbf{R}$ to $\mathbf{R}/\mathbf{S}$, the position of this datum in q, and the first empty position in the window of q_2', respectively. Initially, or when medium $\mathbf{K}$ is inactive, g, e, h and h_2' have the values 5, d_0, 0 and 0. Likewise, g' captures the five states of medium $\mathbf{L}$, and e_2, h_2 and h' represent the memory of $\mathbf{L}$.

The linear specification $\mathbf{M}_{mod}$ of the SWP, with encapsulation but without hiding, takes the following form. For the sake of presentation, in states that results after a transition we only present parameters whose values have changed.

$$\begin{array}{ll}
 & \mathbf{M}_{mod}(\ell, m, q, q_2', \ell_2', g, h, e, h_2', g', h_2, e_2, h', \ell_2, m_2, q_2, q', \ell') \\
\approx & \sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot\mathbf{M}_{mod}(m{:=}S(m)|_{2n}, q{:=}add(d,m,q)) \lhd \; in\text{-}window(\ell, m, (\ell+n)|_{2n}) \rhd \delta \\
+ & \sum_{k:Nat} c_{\mathrm{B}}(retrieve(k,q), k, next\text{-}empty|_{2n_2}(\ell_2', q_2'))\cdot\mathbf{M}_{mod}(g{:=}4, e{:=}retrieve(k,q), h{:=}k, \\
 & \qquad h_2'{:=}next\text{-}empty|_{2n_2}(\ell_2', q_2')) \lhd \; test(k,q) \wedge g=5 \rhd \delta \\
+ & j\cdot\mathbf{M}_{mod}(g{:=}1, e{:=}d_0, h{:=}0) \lhd \; g=2 \rhd \delta
\end{array}$$

$$
\begin{array}{ll}
+ & j{\cdot}\mathbf{M}_{mod}(g{:=}5, e{:=}d_0, h{:=}0, h_2{:=}0) \lhd\ g = 2 \vee g = 4 \rhd \delta \\
+ & j{\cdot}\mathbf{M}_{mod}(g{:=}3) \lhd\ g = 4 \rhd \delta \\
+ & c_{\mathrm{C}}(e, h, h_2'){\cdot}\mathbf{M}_{mod}(\ell_2{:=}h_2', q'{:=}add(e, h, q'), g{:=}5, e{:=}d_0, h{:=}0, h_2'{:=}0, \\
& \qquad q_2{:=}release|_{2n_2}(\ell_2, h_2', q_2)) \lhd\ in\text{-}window(\ell', h, (\ell' + n)|_{2n}) \wedge g = 3 \rhd \delta \\
+ & c_{\mathrm{C}}(e, h, h_2'){\cdot}\mathbf{M}_{mod}(\ell_2{:=}h_2', g{:=}5, e{:=}d_0, h{:=}0, h_2'{:=}0, q_2{:=}release|_{2n_2}(\ell_2, h_2', q_2)) \\
& \qquad \lhd \neg in\text{-}window(\ell', h, (\ell' + n)|_{2n}) \wedge g = 3 \rhd \delta \\
+ & s_{\mathrm{D}}(retrieve(\ell', q')){\cdot}\mathbf{M}_{mod}(\ell'{:=}S(\ell')|_{2n}, q'{:=}remove(\ell', q')) \lhd\ test(\ell', q') \rhd \delta \\
+ & c_{\mathrm{E}}(next\text{-}empty|_{2n}(\ell', q')){\cdot}\mathbf{M}_{mod}(g'{:=}2, h_2{:=}0, h'{:=}next\text{-}empty|_{2n}(\ell', q')) \lhd\ g' = 5 \rhd \delta \\
+ & j{\cdot}\mathbf{M}_{mod}(g'{:=}1, e_2{:=}d_0, h_2{:=}0) \lhd\ g' = 2 \rhd \delta \\
+ & j{\cdot}\mathbf{M}_{mod}(g'{:=}5, h_2{:=}0, e_2{:=}d_0, h'{:=}0) \lhd\ g' = 2 \vee g' = 4 \rhd \delta \\
+ & j{\cdot}\mathbf{M}_{mod}(g'{:=}3) \lhd\ g' = 4 \rhd \delta \\
+ & c_{\mathrm{F}}(h'){\cdot}\mathbf{M}_{mod}(\ell{:=}h', q{:=}release|_{2n}(\ell, h', q), g'{:=}5, h_2{:=}0, e_2{:=}d_0, h'{:=}0) \lhd\ g' = 1 \rhd \delta \\
+ & \sum_{d:\Delta} r_{\mathrm{D}}(d){\cdot}\mathbf{M}_{mod}(m_2{:=}S(m_2)|_{2n}, q_2{:=}add(d, m_2, q_2)) \\
& \qquad \lhd\ in\text{-}window(\ell_2, m_2, (\ell_2 + n_2)|_{2n_2}) \rhd \delta \\
+ & \sum_{k:Nat} c_{\mathrm{E}}(retrieve(k, q_2), k, next\text{-}empty|_{2n}(\ell', q')){\cdot}\mathbf{M}_{mod}(g'{:=}4, e_2{:=}retrieve(k, q_2), \\
& \qquad h_2{:=}k, h'{:=}next\text{-}empty|_{2n}(\ell', q')) \lhd\ test(k, q_2) \wedge g' = 5 \rhd \delta \\
+ & c_{\mathrm{F}}(e_2, h_2, h'){\cdot}\mathbf{M}_{mod}(\ell{:=}h', q_2'{:=}add(e_2, h_2, q_2'), g'{:=}5, e_2{:=}d_0, h_2{:=}0, h'{:=}0, \\
& \qquad q{:=}release|_{2n}(\ell, h', q)) \lhd\ in\text{-}window(\ell_2', h_2, (\ell_2' + n_2)|_{2n_2}) \wedge g' = 3 \rhd \delta \\
+ & c_{\mathrm{F}}(e_2, h_2, h'){\cdot}\mathbf{M}_{mod}(\ell{:=}h', g'{:=}5, e_2{:=}d_0, h_2{:=}0, h'{:=}0, q{:=}release|_{2n}(\ell, h', q)) \\
& \qquad \lhd \neg in\text{-}window(\ell_2', h_2, (\ell_2' + n_2)|_{2n_2}) \wedge g' = 3 \rhd \delta \\
+ & s_{\mathrm{A}}(retrieve(\ell_2', q_2')){\cdot}\mathbf{M}_{mod}(\ell_2'{:=}S(\ell_2')|_{2n_2}, q_2'{:=}remove(\ell_2', q_2')) \lhd\ test(\ell_2', q_2') \rhd \delta \\
+ & c_{\mathrm{B}}(next\text{-}empty|_{2n_2}(\ell_2', q_2')){\cdot}\mathbf{M}_{mod}(g{:=}2, h{:=}0, h_2'{:=}next\text{-}empty|_{2n_2}(\ell_2', q_2')) \lhd\ g = 5 \rhd \delta \\
+ & c_{\mathrm{C}}(h_2'){\cdot}\mathbf{M}_{mod}(\ell_2{:=}h_2', q_2{:=}release|_{2n_2}(\ell_2, h_2', q_2), g{:=}5, h{:=}0, e{:=}d_0, h_2'{:=}0) \lhd\ g = 1 \rhd \delta
\end{array}
$$

The intuition for the LPE $\mathbf{M}_{mod}$ is as follows:

- The first summand describes that a datum d can be received by $\mathbf{S}/\mathbf{R}$ through channel A, if q's window is not full ($in\text{-}window(\ell, m, (\ell + n)|_{2n})$). This datum is then placed in the first empty cell of q's window ($q{:=}add(d, m, q)$), and the next cell becomes the first empty cell of this window ($m{:=}S(m)|_{2n}$).

- By the second summand, a frame ($retrieve(k, q), k, next\text{-}empty|_{2n_2}(\ell_2', q_2')$) can be communicated to $\mathbf{K}$, if cell k in q's window is occupied ($test(k, q)$). And by the nineteenth summand, an acknowledgement $next\text{-}empty|_{2n_2}(\ell_2', q_2')$ can be communicated to $\mathbf{K}$.

- The fifth and third summand describe that medium **K** decides to pass on a frame or acknowledgement, respectively. The fourth summand describes that **K** decides to lose this frame or acknowledgement.

- The sixth and seventh summand describe that the frame in medium **K** is communicated to **R**/**S**. In the sixth summand the frame is within the window of q' ($\mathit{in\text{-}window}(\ell', h, (\ell'+n)|_{2n})$), so it is included ($q'{:=}\mathit{add}(e, h, q')$). In the seventh summand the frame is outside the window of q', so it is omitted. In both cases, the first cell of the window of q' is moved forward to h'_2 ($\ell_2{:=}h'_2$), and the cells before h'_2 are emptied ($q_2{:=}\mathit{release}|_{2n_2}(\ell_2, h'_2, q_2)$).

- The twentieth and last summand describes that the acknowledgement in medium **K** is communicated to **R**/**S**. Then the first cell of the window of q' is moved forward to h'_2, and the cells before h'_2 are emptied.

- By the eighth summand, **R**/**S** can send the datum at the first cell in the window of q' ($\mathit{retrieve}(\ell', q')$) through channel D, if this cell is occupied ($\mathit{test}(\ell', q')$). This cell is then emptied ($q'{:=}\mathit{remove}(\ell', q')$), and the first cell of the window of q' is moved forward by one ($\ell'{:=}S(\ell')|_{2n}$).

The other ten summands are symmetric counterparts to the ones described above.

According to the linearisation algorithm of Groote, Ponse and Usenko [GPU01], we have the following result.

Proposition 6.11 $\partial_{\mathcal{H}}(\mathbf{S}/\mathbf{R}(0,0,[]) \parallel \mathbf{R}/\mathbf{S}(0,[]) \parallel \mathbf{K} \parallel \mathbf{L}) \;\underline{\leftrightarrow}$
$\mathbf{M}_{mod}(0,0,[],[],0,5,0,d_0,0,5,0,d_0,0,0,0,[],[],0)$.

6.5.2 Eliminating Arguments of Communication Actions

The linear specification $\mathbf{N}_{mod}$ is obtained from $\mathbf{M}_{mod}$ by stripping all arguments from communication actions, and renaming these actions to a fresh action c. Sine we want to show that the "external" behaviour of this protocol is branching bisimilar to a pair of FIFO queues (of capacity $2n$), hence the internal actions can be stripped out.

$$
\begin{aligned}
& \mathbf{N}_{mod}(\ell, m, q, q'_2, \ell'_2, g, h, e, h'_2, g', h_2, e_2, h', \ell_2, m_2, q_2, q', \ell') \\
\approx\; & \textstyle\sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot\mathbf{N}_{mod}(m{:=}S(m)|_{2n}, q{:=}\mathit{add}(d,m,q)) \lhd \mathit{in\text{-}window}(\ell, m, (\ell+n)|_{2n}) \rhd \delta \\
+\; & \textstyle\sum_{k:Nat} c\cdot\mathbf{N}_{mod}(g{:=}4, e{:=}\mathit{retrieve}(k,q), h{:=}k, h'_2{:=}\mathit{next\text{-}empty}|_{2n_2}(\ell'_2, q'_2)) \\
& \qquad\qquad \lhd \mathit{test}(k,q) \wedge g = 5 \rhd \delta
\end{aligned}
$$

$+\quad j{\cdot}\mathbf{N}_{mod}(g{:=}1, e{:=}d_0, h{:=}0) \lhd g = 2 \rhd \delta$

$+\quad j{\cdot}\mathbf{N}_{mod}(g{:=}5, e{:=}d_0, h{:=}0, h_2{:=}0) \lhd g = 2 \vee g = 4 \rhd \delta$

$+\quad j{\cdot}\mathbf{N}_{mod}(g{:=}3) \lhd g = 4 \rhd \delta$

$+\quad c{\cdot}\mathbf{N}_{mod}(\ell_2{:=}h'_2, q'{:=}add(e, h, q'), g{:=}5, e{:=}d_0, h{:=}0, h'_2{:=}0, q_2{:=}release|_{2n_2}(\ell_2, h'_2, q_2))$
$\qquad\qquad \lhd in\text{-}window(\ell', h, (\ell' + n)|_{2n}) \wedge g = 3 \rhd \delta$

$+\quad c{\cdot}\mathbf{N}_{mod}(\ell_2{:=}h'_2, g{:=}5, e{:=}d_0, h{:=}0, h'_2{:=}0, q_2{:=}release|_{2n_2}(\ell_2, h'_2, q_2))$
$\qquad\qquad \lhd \neg in\text{-}window(\ell', h, (\ell' + n)|_{2n}) \wedge g = 3 \rhd \delta$

$+\quad s_{\mathrm{D}}(retrieve(\ell', q')){\cdot}\mathbf{N}_{mod}(\ell'{:=}S(\ell')|_{2n}, q'{:=}remove(\ell', q')) \lhd test(\ell', q') \rhd \delta$

$+\quad c{\cdot}\mathbf{N}_{mod}(g'{:=}2, h_2{:=}0, h'{:=}next\text{-}empty|_{2n}(\ell', q')) \lhd g' = 5 \rhd \delta$

$+\quad j{\cdot}\mathbf{N}_{mod}(g'{:=}1, e_2{:=}d_0, h_2{:=}0) \lhd g' = 2 \rhd \delta$

$+\quad j{\cdot}\mathbf{N}_{mod}(g'{:=}5, h_2{:=}0, e_2{:=}d_0, h'{:=}0) \lhd g' = 2 \vee g' = 4 \rhd \delta$

$+\quad j{\cdot}\mathbf{N}_{mod}(g'{:=}3) \lhd g' = 4 \rhd \delta$

$+\quad c{\cdot}\mathbf{N}_{mod}(\ell{:=}h', q{:=}release|_{2n}(\ell, h', q), g'{:=}5, h_2{:=}0, e_2{:=}d_0, h'{:=}0) \lhd g' = 1 \rhd \delta$

$+\quad \sum_{d:\Delta} r_{\mathrm{D}}(d){\cdot}\mathbf{N}_{mod}(m_2{:=}S(m_2)|_{2n}, q_2{:=}add(d, m_2, q_2))$
$\qquad\qquad \lhd in\text{-}window(\ell_2, m_2, (\ell_2 + n_2)|_{2n_2}) \rhd \delta$

$+\quad \sum_{k:Nat} c{\cdot}\mathbf{N}_{mod}(g'{:=}4, e_2{:=}retrieve(k, q_2), h_2{:=}k, h'{:=}next\text{-}empty|_{2n}(\ell', q'))$
$\qquad\qquad \lhd test(k, q_2) \wedge g' = 5 \rhd \delta$

$+\quad c{\cdot}\mathbf{N}_{mod}(\ell{:=}h', q'_2{:=}add(e_2, h_2, q'_2), g'{:=}5, e_2{:=}d_0, h_2{:=}0, h'{:=}0, q{:=}release|_{2n}(\ell, h', q))$
$\qquad\qquad \lhd in\text{-}window(\ell'_2, h_2, (\ell'_2 + n_2)|_{2n_2}) \wedge g' = 3 \rhd \delta$

$+\quad c{\cdot}\mathbf{N}_{mod}(\ell{:=}h', g'{:=}5, e_2{:=}d_0, h_2{:=}0, h'{:=}0, q{:=}release|_{2n}(\ell, h', q))$
$\qquad\qquad \lhd \neg in\text{-}window(\ell'_2, h_2, (\ell'_2 + n_2)|_{2n_2}) \wedge g' = 3 \rhd \delta$

$+\quad s_{\mathrm{A}}(retrieve(\ell'_2, q'_2)){\cdot}\mathbf{N}_{mod}(\ell'_2{:=}S(\ell'_2)|_{2n_2}, q'_2{:=}remove(\ell'_2, q'_2)) \lhd test(\ell'_2, q'_2) \rhd \delta$

$+\quad c{\cdot}\mathbf{N}_{mod}(g{:=}2, h{:=}0, h'_2{:=}next\text{-}empty|_{2n_2}(\ell'_2, q'_2)) \lhd g = 5 \rhd \delta$

$+\quad c{\cdot}\mathbf{N}_{mod}(\ell_2{:=}h'_2, q_2{:=}release|_{2n_2}(\ell_2, h'_2, q_2), g{:=}5, h{:=}0, e{:=}d_0, h'_2{:=}0) \lhd g = 1 \rhd \delta$

The following proposition is trivial.

Proposition 6.12 $\tau_{\mathcal{I}}(\mathbf{M}_{mod}(0, 0, [], [], 0, 5, 0, d_0, 0, 5, 0, d_0, 0, 0, 0, [], [], 0)) \underline{\leftrightarrow}$
$\tau_{\{c\}}(\mathbf{N}_{mod}(0, 0, [], [], 0, 5, 0, d_0, 0, 5, 0, d_0, 0, 0, 0, [], [], 0)).$

6.5.3 Getting Rid of Modulo Arithmetic

The specification of $\mathbf{N}_{nonmod}$ is obtained by eliminating all occurrences of $|_{2k}$ from $\mathbf{N}_{mod}$, and replacing $in\text{-}window(i, j, (i + k)|_{2k})$ by $i \leq j < i + k$.

$$\mathbf{N}_{nonmod}(\ell, m, q, q'_2, \ell'_2, g, h, e, h'_2, g', h_2, e_2, h', \ell_2, m_2, q_2, q', \ell') \approx$$

$$
\begin{aligned}
& \textstyle\sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot\mathbf{N}_{nonmod}(m{:=}S(m), q{:=}add(d,m,q)) \lhd m < \ell + n \rhd \delta && (A)\\
+\; & \textstyle\sum_{k:Nat} c\cdot\mathbf{N}_{nonmod}(g{:=}4, e{:=}retrieve(k,q), h{:=}k, h'_2{:=}next\text{-}empty(\ell'_2, q'_2)) \\
& \qquad \lhd test(k,q) \wedge g = 5 \rhd \delta && (B)\\
+\; & j\cdot\mathbf{N}_{nonmod}(g{:=}1, e{:=}d_0, h{:=}0) \lhd g = 2 \rhd \delta && (C)\\
+\; & j\cdot\mathbf{N}_{nonmod}(g{:=}5, e{:=}d_0, h{:=}0, h_2{:=}0) \lhd g = 2 \vee g = 4 \rhd \delta && (D)\\
+\; & j\cdot\mathbf{N}_{nonmod}(g{:=}3) \lhd g = 4 \rhd \delta && (E)\\
+\; & c\cdot\mathbf{N}_{nonmod}(\ell_2{:=}h'_2, q'{:=}add(e,h,q'), g{:=}5, e{:=}d_0, h{:=}0, h'_2{:=}0, q_2{:=}release(\ell_2, h'_2, q_2)) \\
& \qquad \lhd \ell' \leq h < \ell' + n \wedge g = 3 \rhd \delta && (F)\\
+\; & c\cdot\mathbf{N}_{nonmod}(\ell_2{:=}h'_2, g{:=}5, e{:=}d_0, h{:=}0, h'_2{:=}0, q_2{:=}release(\ell_2, h'_2, q_2)) \\
& \qquad \lhd \neg(\ell' \leq h < \ell' + n) \wedge g = 3 \rhd \delta && (G)\\
+\; & s_{\mathrm{D}}(retrieve(\ell', q'))\cdot\mathbf{N}_{nonmod}(\ell'{:=}S(\ell'), q'{:=}remove(\ell', q')) \lhd test(\ell', q') \rhd \delta && (H)\\
+\; & c\cdot\mathbf{N}_{nonmod}(g'{:=}2, h_2{:=}0, h'{:=}next\text{-}empty(\ell', q')) \lhd g' = 5 \rhd \delta && (I)\\
+\; & j\cdot\mathbf{N}_{nonmod}(g'{:=}1, e_2{:=}d_0, h_2{:=}0) \lhd g' = 2 \rhd \delta && (J)\\
+\; & j\cdot\mathbf{N}_{nonmod}(g'{:=}5, h_2{:=}0, e_2{:=}d_0, h'{:=}0) \lhd g' = 2 \vee g' = 4 \rhd \delta && (K)\\
+\; & j\cdot\mathbf{N}_{nonmod}(g'{:=}3) \lhd g' = 4 \rhd \delta && (L)\\
+\; & c\cdot\mathbf{N}_{nonmod}(\ell{:=}h', q{:=}release(\ell, h', q), g'{:=}5, h_2{:=}0, e_2{:=}d_0, h'{:=}0) \lhd g' = 1 \rhd \delta && (M)\\
+\; & \textstyle\sum_{d:\Delta} r_{\mathrm{D}}(d)\cdot\mathbf{N}_{nonmod}(m_2{:=}S(m_2), q_2{:=}add(d, m_2, q_2)) \lhd m_2 < \ell_2 + n_2 \rhd \delta && (N)\\
+\; & \textstyle\sum_{k:Nat} c\cdot\mathbf{N}_{nonmod}(g'{:=}4, e_2{:=}retrieve(k, q_2), h_2{:=}k, h'{:=}next\text{-}empty(\ell', q')) \\
& \qquad \lhd test(k, q_2) \wedge g' = 5 \rhd \delta && (O)\\
+\; & c\cdot\mathbf{N}_{nonmod}(\ell{:=}h', q'_2{:=}add(e_2, h_2, q'_2), g'{:=}5, e_2{:=}d_0, h_2{:=}0, h'{:=}0, q{:=}release(\ell, h', q)) \\
& \qquad \lhd \ell'_2 \leq h_2 < \ell'_2 + n_2 \wedge g' = 3 \rhd \delta && (P)\\
+\; & c\cdot\mathbf{N}_{nonmod}(\ell{:=}h', g'{:=}5, e_2{:=}d_0, h_2{:=}0, h'{:=}0, q{:=}release(\ell, h', q)) \\
& \qquad \lhd \neg(\ell'_2 \leq h_2 < \ell'_2 + n_2) \wedge g' = 3 \rhd \delta && (Q)\\
+\; & s_{\mathrm{A}}(retrieve(\ell'_2, q'_2))\cdot\mathbf{N}_{nonmod}(\ell'_2{:=}S(\ell'_2), q'_2{:=}remove(\ell'_2, q'_2)) \lhd test(\ell'_2, q'_2) \rhd \delta && (R)\\
+\; & c\cdot\mathbf{N}_{nonmod}(g{:=}2, h{:=}0, h'_2{:=}next\text{-}empty(\ell'_2, q'_2)) \lhd g = 5 \rhd \delta && (S)\\
+\; & c\cdot\mathbf{N}_{nonmod}(\ell_2{:=}h'_2, q_2{:=}release(\ell_2, h'_2, q_2), g{:=}5, h{:=}0, e{:=}d_0, h'_2{:=}0) \lhd g = 1 \rhd \delta && (T)
\end{aligned}
$$

In Section 6.7.1, we will prove that $\mathbf{N}_{nonmod}$ and $\mathbf{N}_{mod}$ are strongly bisimilar. Next, in Section 6.7.2, we will prove the correctness of $\mathbf{N}_{nonmod}$.

In these proofs we will need a wide range of data equalities, which we now proceed to present in Section 6.6.

6.6 Properties of Data Types

This section presents properties of the data types and invariants of the transformed specification; their proofs can be found in the appendix.

6.6.1 Basic Properties

We first list some basic properties for the data types. Most of these lemmas were already presented and proved in [BFG+05]. The first lemma deals with modulo arithmetic.

Unless stated otherwise, all variables that occur in a data lemma are implicitly universally quantified at the outside of the lemma. i, j, k, ℓ, n range over *Nat*, where $n > 0$, q ranges over *Buf*, $\lambda, \lambda', \lambda''$ over *List*, and d over Δ.

Lemma 6.13 *1.* $(i|_n + j)|_n = (i + j)|_n$

2. $i|_n < n$

3. $i = (i \; div \; n) \cdot n + i|_n$

4. $(i \leq j \leq i + n \; \wedge \; i \leq k \leq i + n \; \wedge \; j|_{2n} = k|_{2n}) \; \rightarrow \; j = k$

The next lemma deals with basic properties of buffers.

Lemma 6.14 *1.* $test(i, q) \; \rightarrow \; i \leq max(q)$

2. $\neg test(i, q) \; \rightarrow \; remove(i, q) = q$

3. $test(i, remove(j, q)) = (test(i, q) \wedge i \neq j)$

4. $i \neq j \; \rightarrow \; retrieve(i, remove(j, q)) = retrieve(i, q)$

5. $test(i, release(j, k, q)) = (test(i, q) \wedge \neg(j \leq i < k))$

6. $\neg(j \leq i < k) \; \rightarrow \; retrieve(i, release(j, k, q)) = retrieve(i, q)$

7. $remove(i, remove(j, q)) = remove(j, remove(i, q))$

The next lemma deals with the *next-empty* function.

Lemma 6.15 *1.* $i \leq j < next\text{-}empty(i, q) \; \rightarrow \; test(j, q)$

2. $next\text{-}empty(i, q) \geq i$

3. $\neg(i \leq j < \mathit{next\text{-}empty}(i, q)) \;\rightarrow\; \mathit{next\text{-}empty}(i, \mathit{remove}(j, q)) = \mathit{next\text{-}empty}(i, q)$

The next lemma deals with modulo arithmetic for buffers.

Lemma 6.16 1. $\mathit{next\text{-}empty}|_{2n}(i, q) = \mathit{if}(\mathit{test}(i|_{2n}, q),\ \mathit{next\text{-}empty}|_{2n}(S(i)|_{2n}, q),\ i|_{2n})$

2. $\forall j{:}\mathit{Nat}(\mathit{test}(j, q) \rightarrow i \leq j < i+n) \wedge i \leq k \leq i+n \;\rightarrow\; \mathit{test}(k, q) = \mathit{test}(k|_{2n}, q|_{2n})$

3. $\forall j{:}\mathit{Nat}(\mathit{test}(j, q) \rightarrow i \leq j < i{+}n) \wedge \mathit{test}(k, q) \;\rightarrow\; \mathit{retrieve}(k, q) = \mathit{retrieve}(k|_{2n}, q|_{2n})$

4. $\forall j{:}\mathit{Nat}(\mathit{test}(j, q) \rightarrow i \leq j < i+n) \wedge i \leq k \leq i+n \;\rightarrow$
$\mathit{next\text{-}empty}(k, q)|_{2n} = \mathit{next\text{-}empty}|_{2n}(k|_{2n}, q|_{2n})$

5. $i \leq k < i+n \;\rightarrow\; \mathit{in\text{-}window}(i|_{2n}, k|_{2n}, (i+n)|_{2n})$

6. $\mathit{in\text{-}window}(i|_{2n}, k|_{2n}, (i+n)|_{2n}) \;\rightarrow\; k+n < i \vee i \leq k < i+n \vee k \geq i+2n$

The next lemma presents basic properties of lists.

Lemma 6.17 1. $(\lambda {+\!\!+} \lambda') {+\!\!+} \lambda'' = \lambda {+\!\!+} (\lambda' {+\!\!+} \lambda'')$

2. $\mathit{length}(\lambda {+\!\!+} \lambda') = \mathit{length}(\lambda) + \mathit{length}(\lambda')$

3. $\mathit{append}(d, \lambda {+\!\!+} \lambda') = \lambda {+\!\!+} \mathit{append}(d, \lambda')$

4. $\mathit{length}(q[i..j\rangle) = j \dot{-} i$

5. $i \leq k \leq j \;\rightarrow\; q[i..j\rangle = q[i..k\rangle {+\!\!+} q[k..j\rangle$

6. $\neg(i \leq k < j) \;\rightarrow\; \mathit{remove}(k, q)[i..j\rangle = q[i..j\rangle$

7. $\ell \leq i \;\rightarrow\; \mathit{release}(k, \ell, q)[i..j\rangle = q[i..j\rangle$

8. $i \leq j \;\rightarrow\; \mathit{append}(d, q[i..j\rangle) = \mathit{add}(d, j, q)[i..S(j)\rangle$

9. $\mathit{test}(k, q) \;\rightarrow\; \mathit{add}(\mathit{retrieve}(k, q), k, q)[i..j\rangle = q[i..j\rangle$

6.6.2 Ordered Buffers

In this section we present properties of buffers with ordered sequence numbers without duplication. The first lemma contains some facts on the *add* function.

Lemma 6.18 1. $\mathit{test}(i, q) \;\rightarrow\; \mathit{test}(i, \mathit{add}(d, j, q))$

2. $\mathit{next\text{-}empty}(i, \mathit{add}(d, j, q)) \geq \mathit{next\text{-}empty}(i, q)$

3. $\mathit{test}(i, \mathit{add}(d, j, q)) = (i{=}j \vee \mathit{test}(i, q))$

4. $\mathit{retrieve}(i, \mathit{add}(d, j, q)) = \mathit{if}(i{=}j, d, \mathit{retrieve}(i, q))$

5. $remove(i, add(d, i, q)) = remove(i, q)$

6. $j \neq next\text{-}empty(i, q) \rightarrow next\text{-}empty(i, add(d, j, q)) = next\text{-}empty(i, q)$

7. $next\text{-}empty(i, add(d, next\text{-}empty(i, q), q)) = next\text{-}empty(S(next\text{-}empty(i, q)), q)$

8. $i < j \ \rightarrow \ remove(i, add(d, j, q)) = add(d, j, remove(i, q))$

9. $i \neq j \ \rightarrow \ add(e, i, add(d, j, q)) = add(d, j, add(e, i, q))$

The next lemma deals with the *smaller* and *sort* functions on buffers.

Lemma 6.19 *1.* $smaller(i, q) \ \rightarrow \ smaller(i, remove(j, q))$

2. $i < j \wedge smaller(i, q) \ \rightarrow \ smaller(i, add(d, j, q))$

3. $smaller(i, q) \ \rightarrow \ remove(i, q) = q$

4. $i < j \wedge smaller(j, q) \ \rightarrow \ smaller(i, q)$

5. $sort(q) \ \rightarrow \ sort(add(d, i, q))$

6. $smaller(i, q) \ \rightarrow \ add(d, i, q) = inb(d, i, q)$

7. $sort(q) \wedge j < i \ \rightarrow \ remove(i, add(d, j, q)) = add(d, j, remove(i, q))$

8. $sort(q) \ \rightarrow \ add(d, i, q) = add(d, i, remove(i, q))$

The next lemma collects facts on $q\|_n$.

Lemma 6.20 *Let* $n > 0$.

1. $sort(q\|_n)$

2. $test(i, q|_n) = test(i, q\|_n)$

3. $retrieve(i|_n, q|_n) = retrieve(i|_n, q\|_n)$

4. $j \neq i \ \rightarrow \ remove(i, add(d, j, q\|_n)) = add(d, j, remove(i, q\|_n))$

5. $\forall j{:}Nat(test(j, q) \rightarrow i \leq j < i + n) \wedge i \leq k \leq i + n \rightarrow$
   $next\text{-}empty|_{2n}(k|_{2n}, q|_{2n}) = next\text{-}empty|_{2n}(k|_{2n}, q\|_{2n})$

6. $\forall j{:}Nat(test(j, q) \rightarrow i \leq j < i + n) \wedge i \leq k \leq i + n \rightarrow remove(k, q)\|_{2n} = remove(k|_{2n}, q\|_{2n})$

7. $\forall j{:}Nat(test(j, q) \rightarrow i \leq j < i + n) \wedge i \leq k \leq i + n \rightarrow release(i, k, q)\|_{2n} = release|_{2n}(i|_{2n}, k|_{2n}, q\|_{2n})$

8. $\forall j{:}Nat(test(j, q) \rightarrow i \leq j < i + n) \wedge i \leq k \leq i + n \rightarrow add(d, k, q)\|_{2n} = add(d, k|_{2n}, q\|_{2n})$

6.6.3 Invariants

Invariants of a system are properties of data that are satisfied throughout the reachable state space of the system (see Definition 6.3). Lemma 6.21 collects 19 invariants of $\mathbf{N}_{nonmod}$ (and their symmetric counterparts). Occurrences of variables $i, j{:}Nat$ in an invariant are always implicitly universally quantified at the outside of the invariant.

Invariants 6, 8, 15 and 17 are only needed in the derivation of other invariants. We provide some intuition for the (first of each pair of) invariants that will be used in the correctness proofs in Section 6.7 and in the derivations of the data lemmas in the appendix. Invariants 4, 11, 12, 13 express that the sending window of **S**/**R** is filled from ℓ up to but not including m, and that it has size n. Invariants 7, 10 express that the receiving window of **R**/**S** starts at ℓ' and stops at $\ell' + n$. Invariant 2 expresses that **S**/**R** cannot receive acknowledgements beyond $\mathit{next\text{-}empty}(\ell', q')$, and Invariant 9 that **R**/**S** cannot receive frames beyond $m \dot{-} 1$. Invariants 16, 18, 19 are based on the fact that the sending window of **S**/**R**, the receiving window of **R**/**S**, and **K** (when active) coincide on occupied cells and frames with the same sequence number. Invariants 1, 3, 5 and 14 give bounds on the parameters h and h' of mediums **K** and **L**.

Lemma 6.21 $\mathbf{N}_{nonmod}(\ell, m, q, q_2', \ell_2', g, h, e, h_2', g', h_2, e_2, h', \ell_2, m_2, q_2, q', \ell')$ *satisfies the following invariants.*

1. $h' \leq \mathit{next\text{-}empty}(\ell', q')$
   $h_2' \leq \mathit{next\text{-}empty}(\ell_2', q_2')$
2. $\ell \leq \mathit{next\text{-}empty}(\ell', q')$
   $\ell_2 \leq \mathit{next\text{-}empty}(\ell_2', q_2')$
3. $g' \neq 5 \;\rightarrow\; \ell \leq h'$
   $g \neq 5 \;\rightarrow\; \ell_2 \leq h_2'$
4. $\mathit{test}(i, q) \;\rightarrow\; i < m$
   $\mathit{test}(i, q_2) \;\rightarrow\; i < m_2$
5. $(g = 3 \vee g = 4) \;\rightarrow\; h < m$
   $(g' = 3 \vee g' = 4) \;\rightarrow\; h_2 < m_2$
6. $\mathit{test}(i, q') \;\rightarrow\; i < m$
   $\mathit{test}(i, q_2') \;\rightarrow\; i < m_2$
7. $\mathit{test}(i, q') \;\rightarrow\; \ell' \leq i < \ell' + n$
   $\mathit{test}(i, q_2') \;\rightarrow\; \ell_2' \leq i < \ell_2' + n_2$
8. $\ell' \leq m$
   $\ell_2' \leq m_2$

9. $\textit{next-empty}(\ell', q') \leq m$
$\textit{next-empty}(\ell'_2, q'_2) \leq m_2$

10. $\textit{next-empty}(\ell', q') \leq \ell' + n$
$\textit{next-empty}(\ell'_2, q'_2) \leq \ell'_2 + n_2$

11. $\textit{test}(i, q) \;\rightarrow\; \ell \leq i$
$\textit{test}(i, q_2) \;\rightarrow\; \ell_2 \leq i$

12. $\ell \leq i < m \;\rightarrow\; \textit{test}(i, q)$
$\ell_2 \leq i < m_2 \;\rightarrow\; \textit{test}(i, q_2)$

13. $m \leq \ell + n$
$m_2 \leq \ell_2 + n_2$

14. $(g = 3 \vee g = 4) \;\rightarrow\; \textit{next-empty}(\ell', q') \leq h + n$
$(g' = 3 \vee g' = 4) \;\rightarrow\; \textit{next-empty}(\ell'_2, q'_2) \leq h_2 + n_2$

15. $\ell' \leq i < h' \;\rightarrow\; \textit{test}(i, q')$
$\ell'_2 \leq i < h'_2 \;\rightarrow\; \textit{test}(i, q'_2)$

16. $(g = 3 \vee g = 4) \wedge \textit{test}(h, q) \;\rightarrow\; \textit{retrieve}(h, q) = e$
$(g' = 3 \vee g' = 4) \wedge \textit{test}(h_2, q_2) \;\rightarrow\; \textit{retrieve}(h_2, q_2) = e_2$

17. $(\textit{test}(i, q) \wedge \textit{test}(i, q')) \;\rightarrow\; \textit{retrieve}(i, q) = \textit{retrieve}(i, q')$
$(\textit{test}(i, q_2) \wedge \textit{test}(i, q'_2)) \;\rightarrow\; \textit{retrieve}(i, q_2) = \textit{retrieve}(i, q'_2)$

18. $((g = 3 \vee g = 4) \wedge \textit{test}(h, q')) \;\rightarrow\; \textit{retrieve}(h, q') = e$
$((g' = 3 \vee g' = 4) \wedge \textit{test}(h_2, q'_2)) \;\rightarrow\; \textit{retrieve}(h_2, q'_2) = e_2$

19. $(\ell \leq i \wedge j \leq \textit{next-empty}(i, q')) \;\rightarrow\; q[i..j\rangle = q'[i..j\rangle$
$(\ell_2 \leq i \wedge j \leq \textit{next-empty}(i, q'_2)) \;\rightarrow\; q_2[i..j\rangle = q'_2[i..j\rangle$

It is not hard to check that these invariant are all satisfied in the initial state $\mathbf{N}_{nonmod}(0, 0, [], [], 0, 5, 0, d_0, 0, 5, 0, d_0, 0, 0, 0, [], [], 0)$. So they are satisfied in all reachable states of $\mathbf{N}_{nonmod}$.

6.7 Correctness of $\mathbf{N}_{mod}$

In Section 6.7.1, we prove that $\mathbf{N}_{mod}$ and $\mathbf{N}_{nonmod}$ are strongly bisimilar. Next, in Section 6.7.2 we prove that $\mathbf{N}_{nonmod}$ behaves like a pair of FIFO queues. Finally, Theorem 6.10, stating the correctness of the two-way SWP, is proved in Section 6.7.3.

6.7.1 Equality of $\mathbf{N}_{mod}$ and $\mathbf{N}_{nonmod}$

Proposition 6.22 $\mathbf{N}_{nonmod}(0, 0, [], [], 0, 5, 0, d_0, 0, 5, 0, d_0, 0, 0, 0, [], [], 0) \underline{\leftrightarrow}$
$\mathbf{N}_{mod}(0, 0, [], [], 0, 5, 0, d_0, 0, 5, 0, d_0, 0, 0, 0, [], [], 0)$.

Proof. By Theorem 6.5, it suffices to define a state mapping ϕ and local mappings ψ_j for $j = 1, 2, \ldots, 20$ that satisfy the bisimulation criteria in Definition 6.4, with respect to the invariants in Lemma 6.21.

Let Ξ abbreviate $Nat \times Nat \times Buf \times Buf \times Nat \times Nat \times Nat \times \Delta \times Nat \times Nat \times Nat \times \Delta \times Nat \times Nat \times Nat \times Buf \times Buf \times Nat$. Moreover, $\xi{:}\Xi$ abbreviates

$$(\ell, m, q, q'_2, \ell'_2, g, h, e, h'_2, g', h_2, e_2, h', \ell_2, m_2, q_2, q', \ell').$$

$\phi : \Xi \to \Xi$ is defined as follows.

$$\begin{aligned}\phi(\xi) \quad = \quad & (\ell|_{2n}, m|_{2n}, q\|_{2n}, q'_2\|_{2n_2}, \ell'_2|_{2n_2}, g, h|_{2n}, e, h'_2|_{2n_2},\\ & g', h_2|_{2n_2}, e_2, h'|_{2n}, \ell_2|_{2n_2}, m_2|_{2n_2}, q_2\|_{2n_2}, q'\|_{2n}, \ell'|_{2n})\end{aligned}$$

Furthermore, $\psi_2 : Nat \to Nat$ maps k to $k|_{2n}$, and $\psi_{15} : Nat \to Nat$ maps k to $k|_{2n_2}$; the other 18 local mappings are simply the identity.

We show that ϕ and the ψ_j satisfy the bisimulation criteria. For each summand, we list (and prove) the non-trivial bisimulation criteria that it induces.

A

- $m < \ell + n \leftrightarrow \textit{in-window}(\ell|_{2n}, m|_{2n}, (\ell|_{2n} + n)|_{2n})$.
  $m < \ell + n \leftrightarrow \ell \leq m < \ell + n$ (Inv. 6.21.2 and 6.21.9) $\to$ $\textit{in-window}(\ell|_{2n}, m|_{2n}, (\ell + n)|_{2n})$ (Lem. 6.16.5).
  Reversely, $\textit{in-window}(\ell|_{2n}, m|_{2n}, (\ell + n)|_{2n}) \to m + n < \ell \vee \ell \leq m < \ell + n \vee m \geq \ell + 2n$ (Lem. 6.16.6) $\leftrightarrow m < \ell + n$ (Inv. 6.21.2, 6.21.9 and 6.21.13).
  Furthermore, by Lemma 6.13.1, $(\ell + n)|_{2n} = (\ell|_{2n} + n)|_{2n}$.
- $S(m)|_{2n} = S(m|_{2n})|_{2n}$.
  This follows from Lemma 6.13.1.
- $\textit{add}(d, m, q)\|_{2n} = \textit{add}(d, m|_{2n}, q\|_{2n})$.
  $\textit{test}(k, q) \to \ell \leq k < m$ by Invariants 6.21.4 and 6.21.11. This together with Invariant 6.21.13 and Lemma 6.20.8 gives us $\textit{add}(d, m, q)\|_{2n} = \textit{add}(d, m|_{2n}, q\|_{2n})$.

B

- $\textit{test}(k, q) \to \textit{retrieve}(k, q) = \textit{retrieve}(k|_{2n}, q\|_{2n})$. This follows from Lemma 6.20.3 and Lemma 6.16.3 together with Invariant 6.21.7.
- $\textit{next-empty}(\ell'_2, q'_2)|_{2n_2} = \textit{next-empty}|_{2n_2}(\ell'_2|_{2n_2}, q'_2\|_{2n_2})$. This follows from Lemma 6.20.5 and Lemma 6.16.4 together with Invariant 6.21.7.

F

- $(\ell' \leq h < \ell' + n) \wedge g = 3 \leftrightarrow \textit{in-window}(\ell'|_{2n}, h|_{2n}, (\ell'|_{2n} + n)|_{2n}) \wedge g = 3$.

Let $g = 3$. By Lemma 6.15.2, $\ell' \leq$ *next-empty*(ℓ', q'), and by Invariant 6.21.14 together with $g = 3$, *next-empty*$(\ell', q') \leq h + n$. Hence, $\ell' \leq h + n$. Furthermore, by Invariant 6.21.5 together with $g = 3$, $h < m$, by Invariant 6.21.13, $m \leq \ell + n$, and by Invariants 6.21.2 and 6.21.10, $\ell \leq \ell' + n$. Hence, $h < \ell' + 2n$. So using Lemmas 6.16.5 and 6.16.6, it follows that $\ell' \leq h < \ell' + n \leftrightarrow$ *in-window*$(\ell'|_{2n}, h|_{2n}, (\ell' + n)|_{2n})$. By Lemma 1.1, $(\ell' + n)|_{2n} = (\ell'|_{2n} + n)|_{2n}$.

- $\ell' \leq h < \ell' + n \rightarrow add(e, h, q')\|_{2n} = add(e, h|_{2n}, q'\|_{2n})$.

  This follows from Invariant 6.21.7 and Lemma 6.20.8.

- $g = 3 \rightarrow release(\ell_2, h'_2, q_2)\|_{2n_2} = release|_{2n_2}(\ell_2|_{2n_2}, h'_2|_{2n_2}, q_2\|_{2n_2})$.

  Let $g = 3$. By Invariant 6.21.3, $\ell_2 \leq h'_2$. By Invariant 6.21.1, Invariant 6.21.9 and Invariant 6.21.13, $h'_2 \leq \ell_2 + n_2$. By Invariant 6.21.11, Invariant 6.21.4 and Invariant 6.21.13, $test(k, q_2) \rightarrow \ell_2 \leq k < \ell_2 + n_2$. Using all these and Lemma 6.20.7, we get $release(\ell_2, h'_2, q_2)\|_{2n_2} = release|_{2n_2}(\ell_2|_{2n_2}, h'_2|_{2n_2}, q_2\|_{2n_2})$.

G

- $\neg(\ell' \leq h < \ell' + n) \wedge g = 3 \leftrightarrow \neg$*in-window*$(\ell'|_{2n}, h|_{2n}, (\ell'|_{2n} + n)|_{2n}) \wedge g = 3$.

  This follows immediately from the first item of [F].

- $g = 3 \rightarrow release(\ell_2, h'_2, q_2)\|_{2n_2} = release|_{2n_2}(\ell_2|_{2n_2}, h'_2|_{2n_2}, q_2\|_{2n_2})$.

  This is identical to the last item of [F].

H

- $test(\ell', q') = test(\ell'|_{2n}, q'\|_{2n})$.

  This follows from Lemma 6.20.2 and Lemma 6.16.2 together with Invariant 6.21.7.

- $test(\ell', q') \rightarrow retrieve(\ell', q') = retrieve(\ell'|_{2n}, q'\|_{2n})$.

  This follows from Lemma 6.20.3 and Lemma 6.16.3 together with Invariant 6.21.7.

- $S(\ell')|_{2n} = S(\ell'|_{2n})|_{2n}$.

  This follows from Lemma 6.13.1.

- $remove(\ell', q')\|_{2n} = remove(\ell'|_{2n}, q'\|_{2n})$.

  This follows from Lemma 6.20.6 together with Invariant 6.21.7.

I

- *next-empty*$(\ell', q')|_{2n} =$ *next-empty*$|_{2n}(\ell'|_{2n}, q'\|_{2n})$.

  This follows from Lemma 6.20.5 and Lemma 6.16.4 together with Invariant 6.21.7.

M

- $g' = 1 \rightarrow release(\ell, h', q)\|_{2n} = release|_{2n}(\ell|_{2n}, h'|_{2n}, q\|_{2n})$.

  Let $g' = 1$. By Invariant 6.21.3 together with $g' = 1$, $\ell \leq h'$. By Invariant 6.21.1, $h' \leq$ *next-empty*(ℓ', q'). By Invariant 6.21.9, *next-empty*$(\ell', q') \leq m$.

By Invariant 6.21.13, $m \leq \ell{+}n$. So $\ell \leq h' \leq \ell{+}n$. Hence, the desired equation follows from Lemma 6.20.7 together with Invariants 6.21.4, 6.21.11 and 6.21.13.

Summands N, O, P, Q, R, S and T are the mirrors of the summands A, B, F, G, H, I and M respectively. ■

6.7.2 Correctness of $\mathbf{N}_{nonmod}$

We prove that $\mathbf{N}_{nonmod}$ is branching bisimilar to the pair of FIFO queues $\mathbf{Z}$ (see Section 6.4.2), using cones and foci (see Theorem 6.7)

The state mapping $\phi : \Xi \to \mathit{List} \times \mathit{List}$, which maps states of $\mathbf{N}_{nonmod}$ to states of $\mathbf{Z}$, is defined by:

$$\phi(\xi) = (\phi_1(m, q, \ell', q'), \phi_2(m_2, q_2, \ell'_2, q'_2))$$

where

$$\phi_1(m, q, \ell', q') \quad = \quad q'[\ell'..\mathit{next\text{-}empty}(\ell', q'))\mathbin{+\!\!+} q[\mathit{next\text{-}empty}(\ell', q')..m\rangle$$

$$\phi_2(m_2, q_2, \ell'_2, q'_2) \quad = \quad q'_2[\ell'_2..\mathit{next\text{-}empty}(\ell'_2, q'_2))\mathbin{+\!\!+} q_2[\mathit{next\text{-}empty}(\ell'_2, q'_2)..m_2\rangle$$

Intuitively, ϕ_1 collects data elements in the sending window of $\mathbf{S/R}$ and the receiving window of $\mathbf{R/S}$, starting at the first cell in the receiving window (i.e., ℓ') until the first empty cell in this window, and then continuing in the sending window until the first empty cell in that window (i.e., m). Likewise, ϕ_2 collects data elements in the sending window of $\mathbf{R/S}$ and the receiving window of $\mathbf{S/R}$.

The focus points are states where in the direction from $\mathbf{S/R}$ to $\mathbf{R/S}$, either the sending window of $\mathbf{S/R}$ is empty (meaning that $\ell = m$), or the receiving window from $\mathbf{R/S}$ is full and all data elements in this receiving window have been acknowledged (meaning that $\ell = \ell' + n$). Likewise for the direction from $\mathbf{R/S}$ to $\mathbf{S/R}$. That is, the focus condition reads

$$FC(\xi) \ := \ (\ell = m \vee \ell = \ell' + n) \wedge (\ell_2 = m_2 \vee \ell_2 = \ell'_2 + n_2)$$

Lemma 6.23 *For each $\xi{:}\Xi$ with $\mathbf{N}_{nonmod}(\xi)$ reachable from the initial state, there is a $\hat{\xi}{:}\Xi$ with $FC(\hat{\xi})$ such that $\mathbf{N}_{nonmod}(\xi) \xrightarrow{c_1} \cdots \xrightarrow{c_n} \mathbf{N}_{nonmod}(\hat{\xi})$, where $c_1, \ldots, c_n \in \mathcal{I}$.*

Proof. We prove that for each $\xi{:}\Xi$ where the invariants in Lemma 6.21 hold, there is a finite sequence of internal actions which ends in a state where $(\ell = m \vee \ell = \ell' + n) \wedge (\ell_2 = m_2 \vee \ell_2 = \ell'_2 + n_2)$.

To start with, we evolve to a state where the first part of the conjunction holds. First we show that from each state where $g \neq 5$, a state with $g = 5$ can be reached by

means of internal actions. Next we show that from each reachable state where $g = 5$, a state ξ_0 with $\ell = m \vee \ell = \ell' + n$ can be reached by means of internal actions.

- Consider a state with $g \neq 5$.

  We argue by a case distinction on the value of g that we can perform internal actions to a state with $g = 5$.
  If $g = 2$, with summand C we can get $g = 1$. Then with summand T we can get $g = 5$.
  If $g = 4$, we can get $g = 3$ with summand E, and then with either summand F or G we can get $g = 5$.

- Consider a reachable state with $g = 5$.

  We prove by induction on $\min\{m, \ell' + n\} - \mathit{next\text{-}empty}(\ell', q')$ that a state ξ_0 with $\ell = m \vee \ell = \ell' + n$ can be reached by a sequence of internal actions. By Invariants 6.21.9 and 6.21.10, $\mathit{next\text{-}empty}(\ell', q') \leq \min\{m, \ell' + n\}$.

  - $\mathit{next\text{-}empty}(\ell', q') = \min\{m, \ell' + n\}$.

    * $g' \neq 5$.
      We argue by a case distinction on the value of g' that we can perform internal actions to a state with $g' = 5$, $g = 5$ and $\mathit{next\text{-}empty}(\ell', q') = \min\{m, \ell' + n\}$.
      If $g' = 2$ or $g' = 4$, with summand K we can get $g' = 5$.
      If $g' = 1$, with summand M we can get $g' = 5$.
      If $g' = 3$, with either summand P or Q we can get $g' = 5$.
      The values of g , ℓ', q' and m remain unchanged during all these transitions. Hence $g = 5$ and $\mathit{next\text{-}empty}(\ell', q') = \min\{m, \ell' + n\}$ still hold.
    * $g' = 5$.
      We argue that we can perform three internal actions to a state where $\ell = \mathit{next\text{-}empty}(\ell', q') = \min\{m, \ell' + n\}$.
      Since $g' = 5$, with summand I we can get to $h' = \mathit{next\text{-}empty}(\ell', q')$ and $g' = 2$. Then with summand J we can get to $g' = 1$, while h', ℓ' and q' remain unchanged. Now with summand M we can get to a state where ℓ is given the value of $h' = \mathit{next\text{-}empty}(\ell', q')$, while ℓ' and q' remain unchanged. Hence $\ell = \min\{m, \ell' + n\}$ by the assumption. Therefore $\ell = m \vee \ell = \ell' + n$.

  - $\mathit{next\text{-}empty}(\ell', q') < \min\{m, \ell' + n\}$.

    By Invariant 6.21.2, $\ell \leq \mathit{next\text{-}empty}(\ell', q')$. Using this, the assumption and Invariant 6.21.12, we have $\mathit{test}(\mathit{next\text{-}empty}(\ell', q'), q)$. Since moreover by assumption $g = 5$, with summand B we can get to a state where

$g = 4$, $e = \mathit{retrieve}(k, q)$ and $h = \mathit{next\text{-}empty}(\ell', q')$. Then with summand E we can get $g = 3$, while all other data parameters remain unchanged. By Lemma 6.15.2, $\ell' \leq \mathit{next\text{-}empty}(\ell', q')$. So by the assumption we can use summand F to go to a state where $g = 5$ and q' changes to $\mathit{add}(e, \mathit{next\text{-}empty}(\ell', q'), q')$. Now

$$\begin{aligned} & \mathit{next\text{-}empty}(\ell', \mathit{add}(e, \mathit{next\text{-}empty}(\ell', q'), q')) \\ =& \mathit{next\text{-}empty}(S(\mathit{next\text{-}empty}(\ell', q')), q') && \text{(Lem. 6.18.7)} \\ >& \mathit{next\text{-}empty}(\ell', q') && \text{(Lem. 6.15.2)} \end{aligned}$$

In all the transitions above, ℓ' and m remain unchanged. Moreover, no elements were removed from q', so that $\mathit{next\text{-}empty}(\ell', q')$ did not decrease. Therefore we can apply the induction hypothesis to conclude that we can reach a state ξ_0 with $\ell = m \vee \ell = \ell' + n$ by a sequence of internal actions.

We continue from ξ_0 to reach a focus point $\hat{\xi}$. We need to check that the property $\ell = m \vee \ell = \ell' + n$ remains correct when a transition is performed. Using a similar strategy as in the first part, we show that from each reachable state where $g' \neq 5$ and $\ell = m \vee \ell = \ell' + n$, with a couple of internal actions we can reach a state where $g' = 5$ and $\ell = m \vee \ell = \ell' + n$. Next we show that from each such state a focus point can be reached by a sequence of internal actions.

- Consider a reachable state with $g' \neq 5$ and $\ell = m \vee \ell = \ell' + n$.

  We show how to reach to a state where $g' = 5$ and still $\ell = m \vee \ell = \ell' + n$. With summand K, M, P or Q we can get $g' = 5$. In case of summand K the values of ℓ, m and ℓ' remain the same, but using the other summands ℓ is replaced by h'. Hence it remains to prove that $h' = m \vee h' = \ell' + n$ holds in reachable states with $g' \neq 5$. By Invariants 6.21.1 and 6.21.9, $h' \leq m$. Furthermore, by Invariants 6.21.1 and 6.21.10, $h' \leq \ell' + n$. Hence $h' \leq \min\{m, \ell' + n\}$. On the other hand, by Invariant 6.21.3 and $g' \neq 5$, $\ell \leq h'$. Furthermore, $\ell = m \vee \ell = \ell' + n$ by assumption. Hence $\min\{m, \ell' + n\} \leq h'$. Therefore $h' = \min\{m, \ell' + n\}$. This implies $h' = m \vee h' = \ell' + n$.
- Consider a reachable state with $g' = 5$ and $\ell = m \vee \ell = \ell' + n$.

  We prove by induction on $\min\{m_2, \ell'_2 + n_2\} - \mathit{next\text{-}empty}(\ell'_2, q'_2)$ that a focus point can be reached by a sequence of internal actions. By Invariants 6.21.9 and 6.21.10, $\mathit{next\text{-}empty}(\ell'_2, q'_2) \leq \min\{m_2, \ell'_2 + n_2\}$.

  * $\mathit{next\text{-}empty}(\ell'_2, q'_2) = \min\{m_2, \ell'_2 + n_2\}$.
    · $g \neq 5$.
      With summand D, F, G or T we can go to a state with $g = 5$, $\ell = m \vee \ell = \ell' + n$ and $\mathit{next\text{-}empty}(\ell'_2, q'_2) = \min\{m_2, \ell'_2 + n_2\}$.

· $g = 5$.
We argue that we can perform three internal actions to a state $\hat{\xi}$ where $\ell_2 = \textit{next-empty}(\ell'_2, q'_2) = \min\{m_2, \ell'_2 + n_2\}$. Since $g = 5$, with summand S we can go to a state with $h'_2 = \textit{next-empty}(\ell'_2, q'_2)$ and $g = 2$. Then with summand C we can get $g = 1$, while h'_2, ℓ'_2 and q'_2 remain unchanged. Now with summand T we go to a state where ℓ_2 is given the value of $h'_2 = \textit{next-empty}(\ell'_2, q'_2)$ and ℓ'_2 and q'_2 remain unchanged. Therefore $\ell_2 = \textit{next-empty}(\ell'_2, q'_2) = \min\{m_2, \ell'_2 + n_2\}$. So $\ell_2 = m_2 \vee \ell_2 = \ell'_2 + n_2$. Moreover $\ell = m \vee \ell = \ell' + n$ in $\hat{\xi}$, since ℓ, m and ℓ' remain unchanged during the transitions above. Hence $FC(\hat{\xi})$.

∗ $\textit{next-empty}(\ell'_2, q'_2) < \min\{m_2, \ell'_2 + n_2\}$.
By Invariant6.21.2, $\ell_2 \le \textit{next-empty}(\ell'_2, q'_2)$. So by Invariant 6.21.12 together with the assumption, $\textit{test}(\textit{next-empty}(\ell'_2, q'_2), q_2)$. Since $g' = 5$, with summand O we can go to a state with $g' = 4$, $e_2 = \textit{retrieve}(\textit{next-empty}(\ell'_2, q'_2), q_2)$ and $h_2 = \textit{next-empty}(\ell'_2, q'_2)$. Then with summand L we can get $g' = 3$, while all the other data parameters remain unchanged. By Lemma 6.15.2, $\ell'_2 \le \textit{next-empty}(\ell'_2, q'_2)$. By the assumption we can go with summand P to a state where $g' = 5$, and q'_2 changes to $\textit{add}(e_2, \textit{next-empty}(\ell'_2, q'_2), q'_2)$. Then

$$
\begin{aligned}
&\textit{next-empty}(\ell'_2, \textit{add}(e_2, \textit{next-empty}(\ell'_2, q'_2), q'_2)) && \\
=&\textit{next-empty}(S(\textit{next-empty}(\ell'_2, q'_2)), q'_2) && \text{(Lem. 6.18.7)} \\
>&\textit{next-empty}(\ell'_2, q'_2) && \text{(Lem. 6.15.2)}
\end{aligned}
$$

ℓ'_2 and m_2 remain unchanged through all these transitions, and also ℓ, ℓ' and m did not change. Therefore we can now apply the induction hypothesis to conclude that a focus point $\hat{\xi}$ can be reached by a sequence of internal actions.

■

Proposition 6.24

$$\tau_{\{c\}}(\mathbf{N}_{nonmod}(0, 0, [], [], 0, 5, 0, d_0, 0, 5, 0, d_0, 0, 0, 0, [], [], 0)) \leftrightarrow_b \mathbf{Z}(\langle\rangle, \langle\rangle).$$

Proof. By the cones and foci method (see Theorem 6.7) we obtain the following matching criteria (see Definition 6.6). Trivial matching criteria are left out.

Class I:

1. $\ell' \leq h < \ell' + n \wedge g = 3 \;\rightarrow$
   $\phi(m, q, \ell', q', m_2, q_2, \ell'_2, q'_2) = \phi(m, q, \ell', \mathit{add}(e, h, q'), m_2, \mathit{release}(\ell_2, h'_2, q_2), \ell'_2, q'_2)$
2. $\neg(\ell' \leq h < \ell' + n) \wedge g = 3 \rightarrow \phi_2(m_2, q_2, \ell'_2, q'_2) = \phi_2(m_2, \mathit{release}(\ell_2, h'_2, q_2), \ell'_2, q'_2)$
3. $g' = 1 \;\rightarrow\; \phi_1(m, q, \ell', q') = \phi_1(m, \mathit{release}(\ell, h', q), \ell', q')$
4. $\ell'_2 \leq h_2 < \ell'_2 + n_2 \wedge g' = 3 \;\rightarrow$
   $\phi(m, q, \ell', q', m_2, q_2, \ell'_2, q'_2) = \phi(m, \mathit{release}(\ell, h', q), \ell', q', m_2, q_2, \ell'_2, \mathit{add}(e_2, h_2, q'_2))$
5. $\neg(\ell'_2 \leq h_2 < \ell'_2 + n_2) \wedge g' = 3 \rightarrow \phi_1(m, q, \ell', q') = \phi_1(m, \mathit{release}(\ell, h', q), \ell', q')$
6. $g = 1 \;\rightarrow\; \phi_2(m_2, q_2, \ell'_2, q'_2) = \phi_2(m_2, \mathit{release}(\ell_2, h'_2, q_2), \ell'_2, q'_2)$

Class II:

1. $m < \ell + n \;\rightarrow\; \mathit{length}(\phi_1(m, q, \ell', q')) < 2n$
2. $\mathit{test}(\ell', q') \;\rightarrow\; \mathit{length}(\phi_1(m, q, \ell', q')) > 0$
3. $m_2 < \ell_2 + n_2 \;\rightarrow\; \mathit{length}(\phi_2(m_2, q_2, \ell'_2, q'_2)) < 2n_2$
4. $\mathit{test}(\ell'_2, q'_2) \;\rightarrow\; \mathit{length}(\phi_2(m_2, q_2, \ell'_2, q'_2)) > 0$

Class III:

1. $(\ell = m \vee \ell = \ell' + n) \wedge (\ell_2 = m_2 \vee \ell_2 = \ell'_2 + n_2) \wedge$
   $\mathit{length}(\phi_1(m, q, \ell', q')) < 2n \;\rightarrow\; m < \ell + n$
2. $(\ell = m \vee \ell = \ell' + n) \wedge (\ell_2 = m_2 \vee \ell_2 = \ell'_2 + n_2) \wedge$
   $\mathit{length}(\phi_1(m, q, \ell', q')) > 0 \;\rightarrow\; \mathit{test}(\ell', q')$
3. $(\ell = m \vee \ell = \ell' + n) \wedge (\ell_2 = m_2 \vee \ell_2 = \ell'_2 + n_2) \wedge$
   $\mathit{length}(\phi_2(m_2, q_2, \ell'_2, q'_2)) < 2n_2 \;\rightarrow\; m_2 < \ell_2 + n_2$
4. $(\ell = m \vee \ell = \ell' + n) \wedge (\ell_2 = m_2 \vee \ell_2 = \ell'_2 + n_2) \wedge$
   $\mathit{length}(\phi_2(m_2, q_2, \ell'_2, q'_2)) > 0 \;\rightarrow\; \mathit{test}(\ell'_2, q'_2)$

Class IV:

1. $\mathit{test}(\ell', q') \;\rightarrow\; \mathit{retrieve}(\ell', q') = \mathit{top}(\phi_1(m, q, \ell', q'))$
2. $\mathit{test}(\ell'_2, q'_2) \;\rightarrow\; \mathit{retrieve}(\ell'_2, q'_2) = \mathit{top}(\phi_2(m_2, q_2, \ell'_2, q'_2))$

Class V:

1. $m < \ell + n \;\rightarrow$
   $\phi_1(S(m), add(d,m,q), \ell', q',) = append(d, \phi_1(m,q,\ell',q'))$
2. $test(\ell', q') \;\rightarrow$
   $\phi_1(m, q, S(\ell'), remove(\ell', q')) = tail(\phi_1(m,q,\ell',q'))$
3. $m_2 < \ell_2 + n_2 \;\rightarrow$
   $\phi_2(S(m_2), add(d,m_2,q_2), \ell_2', q_2') = append(d, \phi_2(m_2,q_2,\ell_2',q_2'))$
4. $test(\ell_2', q_2') \;\rightarrow$
   $\phi_2(m_2, q_2, S(\ell_2'), remove(\ell_2', q_2')) = tail(\phi_2(m_2,q_2,\ell_2',q_2'))$

Below we prove that all these matching criteria hold. We only prove one case of each two mirror pairs, since the mirrored one can be proved in a similar fashion.

I.1 $\ell' \leq h < \ell' + n \wedge g = 3 \rightarrow$
$\phi(m,q,\ell',q',m_2,q_2,\ell_2',q_2') = \phi(m,q,\ell', add(e,h,q'), m_2, release(\ell_2, h_2', q_2), \ell_2', q_2')$.

- First we prove $\ell' \leq h < \ell'+n \wedge g = 3 \rightarrow \phi_1(m,q,\ell',q') = \phi_1(m,q,\ell', add(e,h,q'))$.

  Case 1: $h \neq next\text{-}empty(\ell', q')$.
  By Lemma 6.18.6, $next\text{-}empty(\ell', add(e,h,q')) = next\text{-}empty(\ell', q')$. Hence,

  $$add(e,h,q')[\ell'..next\text{-}empty(\ell', add(e,h,q')))\rangle +\!\!+ q[next\text{-}empty(\ell', add(e,h,q'))..m\rangle =$$

  $$add(e,h,q')[\ell'..next\text{-}empty(\ell', q')\rangle +\!\!+ q[next\text{-}empty(\ell', q')..m\rangle.$$

  Case 1.1: $test(h, q')$.
  By Invariant 6.21.18 together with $test(h,q')$ and $g = 3$, $retrieve(h,q') = e$. So by Lemma 6.17.9 and $test(h,q')$, $add(e,h,q')[\ell'..next\text{-}empty(\ell',q')\rangle = q'[\ell'..next\text{-}empty(\ell',q')\rangle$.

  Case 1.2: $\neg test(h,q')$.
  Since $\ell' \leq h$, by Lemma 6.15.1, $next\text{-}empty(\ell',q') \leq h$. Then

  $$\begin{aligned}
  & add(e,h,q')[\ell'..next\text{-}empty(\ell',q')\rangle \\
  &= remove(h, add(e,h,q'))[\ell'..next\text{-}empty(\ell',q')\rangle && \text{(Lem. 6.17.6)}\\
  &= remove(h,q')[\ell'..next\text{-}empty(\ell',q')\rangle && \text{(Lem. 6.18.5)}\\
  &= q'[\ell'..next\text{-}empty(\ell',q')\rangle && \text{(Lem. 6.17.6)}
  \end{aligned}$$

  Case 2: $h = next\text{-}empty(\ell', q')$. We prove the desired equality in three steps:

(1)

$$\begin{aligned}
&\mathit{next\text{-}empty}(\ell', \mathit{add}(e,h,q')) \\
&= \mathit{next\text{-}empty}(\ell', \mathit{add}(e, \mathit{next\text{-}empty}(\ell',q'), q')) \\
&= \mathit{next\text{-}empty}(S(\mathit{next\text{-}empty}(\ell',q')), q') && \text{(Lem. 6.18.7)} \\
&= \mathit{next\text{-}empty}(S(h), q')
\end{aligned}$$

(2)

$$\begin{aligned}
&\mathit{add}(e,h,q')[\ell'..h\rangle \\
&= \mathit{remove}(h, \mathit{add}(e,h,q'))[\ell'..h\rangle && \text{(Lem. 6.17.6)} \\
&= \mathit{remove}(h,q')[\ell'..h\rangle && \text{(Lem. 6.18.5)} \\
&= q'[\ell'..h\rangle && \text{(Lem. 6.17.6)}
\end{aligned}$$

(3) By Invariant 6.21.2, $\ell \leq h$, and by Invariant 6.21.5 together with $g = 3$, $h < m$. Thus, by Invariant 6.21.12, $\mathit{test}(h,q)$. So by Invariant 6.21.16 together with $g = 3$, $\mathit{retrieve}(h,q) = e$. Hence,

$$\begin{aligned}
& \mathit{add}(e,h,q')[h..\mathit{next\text{-}empty}(S(h),q')\rangle \\
= \;& \mathit{inl}(\mathit{retrieve}(h, \mathit{add}(e,h,q')), \\
& \qquad \mathit{add}(e,h,q')[S(h)..\mathit{next\text{-}empty}(S(h),q')\rangle) \\
= \;& \mathit{inl}(e, \mathit{add}(e,h,q')[S(h)..\mathit{next\text{-}empty}(S(h),q')\rangle) && \text{(Lem. 6.18.4)} \\
= \;& \mathit{inl}(e, q'[S(h)..\mathit{next\text{-}empty}(S(h),q')\rangle) && \text{(Lem. 6.17.6)} \\
= \;& \mathit{inl}(e, q[S(h)..\mathit{next\text{-}empty}(S(h),q')\rangle) && \text{(Inv. 6.21.19)} \\
= \;& q[h..\mathit{next\text{-}empty}(S(h),q')\rangle
\end{aligned}$$

Finally, we combine (1), (2), (3). By the assumption and Lemma 6.15.2, $\ell' \leq h < \mathit{next\text{-}empty}(S(h),q')$. Furthermore, by Invariant 6.21.6, $\neg\mathit{test}(m,q')$, and by Invariant 6.21.5 and $g = 3$, $S(h) \leq m$. So in view of Lemma 6.15.1, $\mathit{next\text{-}empty}(S(h),q') \leq m$.

$$\begin{aligned}
& \mathit{add}(e,h,q')[\ell'..\mathit{next\text{-}empty}(\ell', \mathit{add}(e,h,q'))\rangle \\
& +\!\!+ q[\mathit{next\text{-}empty}(\ell', \mathit{add}(e,h,q'))..m\rangle \\
= \;& \mathit{add}(e,h,q')[\ell'..\mathit{next\text{-}empty}(S(h),q')\rangle \\
& +\!\!+ q[\mathit{next\text{-}empty}(S(h),q')..m\rangle && (1) \\
= \;& (\mathit{add}(e,h,q')[\ell'..h\rangle +\!\!+ \mathit{add}(e,h,q')[h..\mathit{next\text{-}empty}(S(h),q')\rangle) \\
& +\!\!+ q[\mathit{next\text{-}empty}(S(h),q')..m\rangle && \text{(Lem. 6.17.5)} \\
= \;& (q'[\ell'..h\rangle +\!\!+ q[h..\mathit{next\text{-}empty}(S(h),q')\rangle) \\
& +\!\!+ q[\mathit{next\text{-}empty}(S(h),q')..m\rangle && (2), (3) \\
= \;& q'[\ell'..h\rangle +\!\!+ q[h..m\rangle && \text{(Lem. 6.17.1, 6.17.5)} \\
= \;& q'[\ell..\mathit{next\text{-}empty}(\ell',q')\rangle +\!\!+ q[\mathit{next\text{-}empty}(\ell',q')..m\rangle
\end{aligned}$$

- Second we prove:
$\ell' \leq h < \ell' + n \wedge g = 3 \rightarrow \phi_2(m_2, q_2, \ell'_2, q'_2) = \phi_2(m_2, \mathit{release}(\ell_2, h'_2, q_2), \ell'_2, q'_2)$.

  By Invariant 6.21.1, $h'_2 \leq \mathit{next\text{-}empty}(\ell'_2, q'_2)$. So by Lemma 6.17.7,

$$\mathit{release}(\ell_2, h'_2, q_2)[\mathit{next\text{-}empty}(\ell'_2, q'_2)..m_2\rangle = q_2[\mathit{next\text{-}empty}(\ell'_2, q'_2)..m_2\rangle$$

I.2 $\neg(\ell' \leq h < \ell' + n) \wedge g = 3 \rightarrow \phi_2(m_2, q_2, \ell'_2, q'_2) = \phi_2(m_2, \mathit{release}(\ell_2, h'_2, q_2), \ell'_2, q'_2)$.

This can be proved in a similar way as the previous case.

I.3 $g' = 1 \rightarrow \phi(m, q, \ell', q', m_2, q_2, \ell'_2, q'_2) = \phi(m, \mathit{release}(\ell, h', q), \ell', q', m_2, q_2, \ell'_2, q'_2)$.
By Invariant 6.21.1, $h' \leq \mathit{next\text{-}empty}(\ell', q')$. So by Lemma 6.17.7

$$\mathit{release}(\ell, h', q)[\mathit{next\text{-}empty}(\ell', q')..m\rangle = q[\mathit{next\text{-}empty}(\ell', q')..m\rangle.$$

II.1 $m < \ell + n \rightarrow \mathit{length}(\phi_1(m, q, \ell', q')) < 2n$.

$$\begin{array}{cll} & \mathit{length}(q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle +\!\!+ q[\mathit{next\text{-}empty}(\ell', q')..m\rangle) & \\ = & \mathit{length}(q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle) + \mathit{length}(q[\mathit{next\text{-}empty}(\ell', q')..m\rangle)) & \text{(Lem. 6.17.2)} \\ = & (\mathit{next\text{-}empty}(\ell', q') \dot{-} \ell') + (m \dot{-} \mathit{next\text{-}empty}(\ell', q')) & \text{(Lem. 6.17.4)} \\ \leq & n + (m \dot{-} \ell) & \text{(Inv. 6.21.2, Inv. 6.21.10)} \\ < & 2n & (m < \ell + n) \end{array}$$

II.2 $\mathit{test}(\ell', q') \rightarrow \mathit{length}(\phi_1(m, q, \ell', q')) > 0$

$\mathit{test}(\ell', q')$ together with Lemma 6.15.2 yields $\mathit{next\text{-}empty}(\ell', q') = \mathit{next\text{-}empty}(S(\ell'), q') \geq S(\ell')$. Hence, by Lemmas 6.17.2 and 6.17.4,

$$\begin{array}{rcl} 0 & < & (\mathit{next\text{-}empty}(\ell', q') \dot{-} \ell') + (m \dot{-} \mathit{next\text{-}empty}(\ell', q')) \\ & = & \mathit{length}(q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle +\!\!+ q[\mathit{next\text{-}empty}(\ell', q')..m\rangle) \end{array}$$

III.1 $(\ell = m \vee \ell = \ell' + n) \wedge (\ell_2 = m_2 \vee \ell_2 = \ell'_2 + n_2) \wedge \mathit{length}(\phi_1(m, q, \ell', q')) < 2n \rightarrow m < \ell + n$

Case 1: $\ell = m$.
Then $m < \ell + n$ holds trivially, since $n > 0$.

Case 2: $\ell = \ell' + n$.

By Invariant 6.21.10, $\mathit{next\text{-}empty}(\ell', q') \leq \ell' + n$. Hence,

$$\begin{array}{lll} & \mathit{length}(q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle +\!\!+ q[\mathit{next\text{-}empty}(\ell', q')..m\rangle) < 2n & \\ \to & (\mathit{next\text{-}empty}(\ell', q') \dot{-} \ell') + (m \dot{-} \mathit{next\text{-}empty}(\ell', q'))) < 2n & \\ \to & m \dot{-} \ell' < 2n & \text{(Lem. 6.15.2)} \\ \to & m < \ell + n & (\ell = \ell' + n) \end{array}$$

III.2 $(\ell = m \vee \ell = \ell' + n) \wedge (\ell_2 = m_2 \vee \ell_2 = \ell'_2 + n_2) \wedge \mathit{length}(\phi_1(m, q, \ell', q')) > 0 \;\to\; \mathit{test}(\ell', q')$.

CASE 1: $\ell = m$.
Then $m \dot{-} \mathit{next\text{-}empty}(\ell', q') \leq (m \dot{-} \ell)$(Inv. 6.21.2) $= 0$, so

$$\begin{array}{lll} 0 & < & \mathit{length}(q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle +\!\!+ q[\mathit{next\text{-}empty}(\ell', q')..m\rangle) \\ & = & \mathit{next\text{-}empty}(\ell', q') \dot{-} \ell' \end{array}$$

Hence $\mathit{next\text{-}empty}(\ell', q') > \ell'$, which implies $\mathit{test}(\ell', q')$.

CASE 2: $\ell = \ell' + n$.
Then by Invariant 6.21.2, $\mathit{next\text{-}empty}(\ell', q') \geq \ell' + n$, which implies $\mathit{test}(\ell', q')$.

IV $\mathit{test}(\ell', q') \;\to\; \mathit{retrieve}(\ell', q') = \mathit{top}(\phi_1(m, q, \ell', q'))$.

Since $\mathit{test}(\ell', q')$,

$$\begin{array}{ll} & \mathit{next\text{-}empty}(\ell', q') \\ = & \mathit{next\text{-}empty}(S(\ell'), q') \\ \geq & S(\ell') \qquad \text{(Lem. 6.15.2)} \end{array}$$

Hence, $q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle = \mathit{inl}(\mathit{retrieve}(\ell', q'), q'[S(\ell')..\mathit{next\text{-}empty}(\ell', q')\rangle)$.
This implies $\mathit{top}(q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle +\!\!+ q[\mathit{next\text{-}empty}(\ell', q')..m\rangle) = \mathit{retrieve}(\ell', q')$.

V.1 $m < \ell + n \;\to\; \phi_1(S(m), \mathit{add}(d, m, q), \ell', q') = \mathit{append}(d, \phi_1(m, q, \ell', q'))$

$$\begin{array}{lll} & q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle +\!\!+ & \\ & \mathit{add}(d, m, q)[\mathit{next\text{-}empty}(\ell', q')..S(m)\rangle & \\ = & q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle +\!\!+ & \\ & \mathit{append}(d, q[\mathit{next\text{-}empty}(\ell', q')..m\rangle) & \text{(Lem. 6.17.8, Inv. 6.21.9)} \\ = & \mathit{append}(d, q'[\ell'..\mathit{next\text{-}empty}(\ell', q')\rangle +\!\!+ & \\ & q[\mathit{next\text{-}empty}(\ell', q')..m\rangle) & \text{(Lem. 6.17.3)} \end{array}$$

V.2 $test(\ell', q') \;\rightarrow\; \phi_1(m, q, S(\ell'), remove(\ell', q')) = tail(\phi_1(m, q, \ell', q'))$.

$$\begin{array}{lll}
 & remove(\ell', q')[S(\ell')..next\text{-}empty(S(\ell'), remove(\ell', q'))\rangle & \\
 & +\!\!+ q[next\text{-}empty(S(\ell'), remove(\ell', q'))..m\rangle & \\
= & remove(\ell', q')[S(\ell')..next\text{-}empty(S(\ell'), q')\rangle +\!\!+ & \\
 & q[next\text{-}empty(S(\ell'), q')..m\rangle & \text{(Lem. 6.15.3)} \\
= & remove(\ell', q')[S(\ell')..next\text{-}empty(\ell', q')\rangle +\!\!+ q[next\text{-}empty(\ell', q')..m\rangle & (test(\ell', q')) \\
= & q'[S(\ell')..next\text{-}empty(\ell', q')\rangle +\!\!+ q[next\text{-}empty(\ell', q')..m\rangle & \text{(Lem. 6.17.6)} \\
= & tail(inl(retrieve(\ell', q'), q'[S(\ell')..next\text{-}empty(\ell', q')\rangle) & \\
 & +\!\!+ q[next\text{-}empty(\ell', q')..m\rangle) & \\
= & tail(q'[\ell'..next\text{-}empty(\ell', q')\rangle +\!\!+ q[next\text{-}empty(\ell', q')..m\rangle) & (test(\ell', q'))
\end{array}$$

■

6.7.3 Correctness of the Two-Way Sliding Window Protocol

Finally, we can prove Theorem 6.10, which states that

$$\tau_{\mathcal{I}}(\partial_{\mathcal{H}}(\mathbf{S}/\mathbf{R}(0, 0, n, n_2, [], [], 0) \parallel \mathbf{R}/\mathbf{S}(0, 0, n, n_2, [], [], 0) \parallel \mathbf{K} \parallel \mathbf{L})) \;\underline{\leftrightarrow}_b\; \mathbf{Z}(\langle\rangle, \langle\rangle)$$

Proof. We combine the equivalences that have been obtained so far:

$$\begin{array}{lll}
 & \tau_I(\partial_H(\mathbf{S}/\mathbf{R}(0, 0, n, n_2, [], [], 0) \parallel \mathbf{K} \parallel \mathbf{R}/\mathbf{S}(0, 0, n, n_2, [], [], 0) \parallel \mathbf{L})) & \\
\underline{\leftrightarrow} & \tau_{\{c,j\}}(\mathbf{M}_{mod}(0, 0, [], [], 0, 5, 0, d_0, 0, 5, 0, d_0, 0, 0, 0, [], [], 0)) & \text{(Proposition 6.11)} \\
\underline{\leftrightarrow} & \tau_{\{c,j\}}(\mathbf{N}_{mod}(0, 0, [], [], 0, 5, 0, d_0, 0, 5, 0, d_0, 0, 0, 0, [], [], 0)) & \text{(Proposition 6.12)} \\
\underline{\leftrightarrow} & \tau_{\{c,j\}}(\mathbf{N}_{nonmod}(0, 0, [], [], 0, 5, 0, d_0, 0, 5, 0, d_0, 0, 0, 0, [], [], 0)) & \text{(Proposition 6.22)} \\
\underline{\leftrightarrow}_b & \mathbf{Z}(\langle\rangle, \langle\rangle) & \text{(Proposition 6.24)}
\end{array}$$

■

7

Formalization and Verification in PVS

In this chapter we show the formalization and verification of the correctness proof of the SWP with piggybacking in PVS [ORR+96].

The PVS specification language is based on simply typed higher-order logic. Its type system contains basic types such as *boolean, nat, integer, real,* etc. and type constructors such as *set, tuple, record,* and *function.* Tuple types have the form `[T1,...,Tn]`, where `Ti` are type expressions. A record is a finite list of fields of the form `R:TYPE=[# E1:T1, ...,En:Tn #]`, where `Ei` are *record accessor* functions. A function type constructor has the form `F:TYPE=[T1,...,Tn->R]`, where `F` is a function with domain `D=T1`$\times$`...`$\times$`Tn` and range `R` [FPvdP05].

A PVS specification can be structured through a hierarchy of *theories.* Each theory consists of a *signature* for the type names and constants introduced in the theory, and a number of axioms, definitions and theorems associated with the signature. A PVS theory can be parametric in certain specified types and values, which are placed between `[ ]` after the theory name.

In μCRL, the semantics of a data specification is the set of all its models. Incomplete data specifications may have multiple models. Even worse, it is possible to have inconsistent data specifications for which no models exist. Here the necessity of specification with PVS emerges, because of this probable incompleteness and inconsistency which exists when working with μCRL. Moreover, PVS was used to search for omissions and errors in the manual μCRL proof of the SWP with piggybacking.

In Section 7.1 we show examples of the original specification of some data functions, then we introduce the modified forms of them. Moreover, we show how measure functions are used to detect the termination of recursive definitions. In Section 7.2 and 7.3 we represent the LPEs and invariants of the SWP with piggybacking in PVS. Section 7.4 presents the equality of μCRL specification of the SWP with piggybacking with and without modulo arithmetic. Section 7.5 explains how the cones and foci

method is used to formalize the main theorem, that is the μCRL specification of the SWP with piggybacking is branching bisimilar to a FIFO queue of size $2n$. Finally, Section 7.6 is dedicated to some remarks on the verification in PVS.

7.1 Data Specifications in PVS

In PVS, all the definitions are first type checked, which generates some *proof obligations*. Proving all these obligations ascertains that our data specification is complete and consistent.

To achieve this, having total definitions is required. So in the first place, partially defined functions need to be extended to total ones. Below there are some examples of partial definitions in the original data specification of the SWP with piggybacking, which we changed into total ones. Second, to guarantee totality of recursive definitions, PVS requires the user to define a so-called *measure function*. Doing this usually requires time and effort, but the advantage is that recursive definitions are guaranteed to be well-founded. PVS enabled us to find non-terminating definitions in the original data specification of the SWP with piggybacking, which were not detected within the framework of μCRL. After finding these non-terminating definitions with PVS, we searched for new definition which can express the operation we look for. Then we replaced the old definitions with new terminating ones in our μCRL framework. Below we show some of the most interesting examples.

Example 7.1 *We defined a function next-empty which gives the first empty position in q from a certain position by: $\mathit{next\text{-}empty}(i,q) = \mathit{if}(\mathit{test}(i,q), \mathit{next\text{-}empty}(S(i),q), i)$. We also need to have $\mathit{next\text{-}empty}|_n(i,q)$ as a function which produces the first empty position in q modulo n, from position i. It looked reasonable to define it as:*

$$\mathit{next\text{-}empty}|_n(i,q) \quad = \quad \mathit{if}(\mathit{test}(i,q), \mathit{next\text{-}empty}|_n(S(i)|_n, q), i)$$

Although the definition looks total and well-founded, this was one of the undetected potential errors that PVS detected during the type checking process. Below we bring an example to show what happens. Let

$$q = [(d_0,0),(d_1,1),(d_2,2),(d_3,3),(d_5,5)],\ n = 4,\ i = 5$$

then

$$\begin{aligned}
\textit{next-empty}|_4(5,q) &= \textit{next-empty}|_4(6|_4,q)\\
&= \textit{next-empty}|_4(2,q)\\
&= \textit{next-empty}|_4(3,q)\\
&= \textit{next-empty}|_4(0,q)\\
&= \textit{next-empty}|_4(1,q)\\
&= \textit{next-empty}|_4(2,q)\\
&= \textit{next-empty}|_4(3,q)\\
&= \ldots
\end{aligned}$$

which will never terminate. The problem is that modulo n all the places in q are occupied, and since $0 \leq i|_n < n$ hence test(i, q) will always be `true`*. Hence each position will call for its immediate next position and so on. Therefore the calls will never stop.*

At the end we replaced it with the following definition, which is terminating and operates the way as we expect.

$$\begin{aligned}
\textit{next-empty}|_n(i,q) \quad = \quad & \textit{if}\,(\textit{next-empty}(i|_n,q) < n, \textit{next-empty}(i|_n,q),\\
& \textit{if}\,(\textit{next-empty}(0,q) < n, \textit{next-empty}(0,q), n))
\end{aligned}$$

This function first checks whether there is any empty place after $i|_n$ (incl. $i|_n$ itself). If this is the case then that position would be the result, otherwise using next-empty$(0, q)$ it will check if there is any empty position in the buffer modulo n. If so then that position would be the value of the function since next-empty$(i|_n, q)$ will reach it. If all the buffer modulo n is full then n would be the result, because n is bigger that all the possible values for the function (i.e. $i|_n$ at most) and moreover it indicates that the buffer is full modulo n.

Example 7.2 *release(i, j, q) is obtained by emptying positions i up to j in q, as it is defined in Section 6.3.3. The original definition was the one below which we modified, because PVS detected non-termination on it.*

$$\textit{release}(i,j,q) \quad = \quad \textit{if}\,(i = j, q, \textit{release}(S(i), j, \textit{remove}(i,q)))$$

It is non-terminating when $i > j$. Therefore we replaced $i = j$ with $i \geq j$ in the case distinction above.

Example 7.3 *release$|_n(i, j, q)$ behaves similar to release(i, j, q) modulo n. The previous error on the release(i, j, q) definition does not apply here, since $i|_n$ will not grow*

beyond $n-1$. First, we defined it as follows:

$$release|_n(i,j,q) \quad = \quad if\,(i=j, q, release(S(i)|_n, j, remove(i,q)))$$

This definition met our expectations, except there was an undetected problem inside of it, that can cause a non-termination. This problem occurs if $i = j+1$ and $j > n$. Thus we modified the above definition to:

$$release|_n(i,j,q) \quad = \quad if\,(i|_n = j|_n, q, release|_n(S(i), j, remove(i|_n, q)))$$

This new definition works properly and is terminating. In Figure 7.1, it is shown how the auxiliary function dm measures this function's reduction, to make sure it is total.

We represented the μCRL abstract data types directly by PVS types. This enables us to reuse the PVS library for definitions and theorems of "standard" data types. As an illustration, Figure 7.1 shows part of a PVS theory defining $release|_n$. There D is an unspecified but non-empty type which represents the set of all datums that can be communicated between the sender and the receiver. `Buf` is list of pairs of type $D \times Nat$ defined as $list[[D, nat]]$. Here we used *list* to identify the type of lists, which is defined in the prelude in PVS. Therefore we simply use it without any need to define it explicitly. This figure also represents $release|_n(i,j,q)$ in PVS. Since it is defined recursively, in order to establish its termination (or totality), it is required by PVS to have a measure function. We define a measure function called *dm* which is decreasing and non-recursive. Here, PVS uses its type-checker to check the validity of *dm*. It generates two type-check proof obligations: if $i|_n < j|_n$ then $j|_n - i|_n \geq 0$ and if $i|_n \geq j|_n$ then $n + j|_n - i|_n \geq 0$. The first proof obligation is proved in one trivial step. The second one is proved by imposing Lemma 6.13.2 on it.

PVS does not allow to skip the proofs of basic properties of the operations on *Nat* and *Bool*, which were mentioned in Section 6.6.1. Below we list all auxiliary lemmas for *Nat* and *Bool* that PVS requires to be defined and proved literally, while in the μCRL proof we considered them as trivial facts. For the proofs, the reader is referred to `http://seshome.informatik.uni-oldenburg.de/~bahareh/piggybacking.dump`.

Lemma 7.4 *The following statements hold for $n > 0$ and $i, j \in Nat$:*

1. $i > 0 \;\rightarrow\; i{\cdot}n \geq n$
2. $i > 0 \;\rightarrow\; i \dot{-} n < i$
3. $i|_n \leq i$
4. $S(i)|_n \leq S(i|_n)$
5. $i|_n \neq n-1 \;\rightarrow\; i|_n < S(i)|_n$

```
...
D:nonempty_type
Buf:type=list[[D,nat]]
x,i,j,k,l,n:  VAR nat
...
dm(i,j,n):  nat =
      IF mod(i,n)<=mod(j,n)
      THEN mod(j,n)-mod(i,n)
      ELSE n+mod(j,n)-mod(i,n)
      ENDIF
...
release(n)(i,j,q):  RECURSIVE Buf=
      IF mod(i,n)=mod(j,n) THEN q
      ELSE release(n)(mod(i+1,n),j,remove(mod(i,n),q))
      ENDIF
      measure dm(i,j,n)
...
```

Figure 7.1: An example of data specification in PVS

6. $i \leq j \ \rightarrow (i \ div \ n) \leq (j \ div \ n)$
7. $i \leq j \leq i+n \ \rightarrow (j \ div \ n) = (i \ div \ n) \vee (j \ div \ n) = S(i \ div \ n)$
8. $test(i, q|_n) \ \rightarrow \ i < n$
9. $i+n \leq j < i+2n \ \rightarrow \ \neg in\text{-}window(i|_{2n}, j|_{2n}, (i+n)|_{2n})$
10. $(q|_n)|_n = q|_n$
11. $\lambda +\!\!+ \langle\rangle = \lambda$
12. $test(i, q) \ \rightarrow \ test(i|_n, q|_n)$

Several data lemmas contain many back and forth steps in their proof strategies in the μCRL proof, which are complicated to be done in PVS, so that some of the proofs have been restructured or modified in PVS in such a way that they can be obtained without any detour. For example, Lemma 6.16.5 is proved by using Lemmas 7.4.6 and 7.4.7 above.

7.2 Representing LPEs

We now reuse [FPvdP05] to show how the μCRL specification of the SWP with piggybacking (an LPE) can be represented in PVS. The main distinction will be that we have assumed so far that LPEs are *clustered*. This means that each action label occurs in at most one summand, so that the set of summands could be indexed by

```
LPE[Act,State,Local:TYPE,n:nat]:  THEORY BEGIN
      SUMMAND:TYPE= [State,Local-> [#act:Act,guard:bool,next:State#] ]
      LPE:TYPE= [#init:State,sums:[below(n)->SUMMAND]#]
END LPE
```

Figure 7.2: Definition of LPE in PVS

the set of action labels. This is no limitation, because any LPE can be transformed in clustered form, basically by replacing $+$ by $\sum$ over finite types. Clustered LPEs enable a notationally smoother presentation of the theory. However, when working with concrete LPEs this restriction is not convenient, so we avoid it in the PVS framework: an arbitrarily sized index set $\{0,\ldots,n-1\}$ will be used, represented by the PVS type `below(n)`. A second deviation is that we will assume from now on that every summand has the same set of local variables. Again this is no limitation, because void summations can always be added (i.e. $p = \sum_{d:D} p$, when d doesn't occur in p). This restriction is needed to avoid the use of polymorphism, which doesn't exist in PVS. The third deviation is that we don't distinguish action labels from action data parameters. We simply work with one type of expressions for actions. Note that this is a real extension, because one summand may now generate steps with various action labels, possibly visible as well as invisible.

So an LPE is parameterized by sets of actions (`Act`), global parameters (`State`) and local variables (`Local`), and by the size of its index set (`n`). Note that the guard, action and next-state of a summand depend on the global parameters $d : State$ and on local variables $e : Local$. This dependency is represented in the definition `SUMMAND` by a PVS function type. An LPE (see Figure 7.2) consists of an initial state and a list of summands indexed by `below(n)`.

Figure 7.3 illustrates the definition of a concrete LPE by a fragment of the linear specification $\mathbf{N}_{mod}$ of SWP with piggybacking in PVS. It is introduced as an `lpe` of a set of actions: `Nnonmod_act`, states: `State`, local variables: `Local`, and a digit: `20` referring to the number of summands. The LPE is identified as a pair, called `init` and `sums`, where `init` is introducing the initial state of $\mathbf{N}_{mod}$ and `sums` the summands. The first `LAMBDA` maps each number to the corresponding summand in $\mathbf{N}_{mod}$. The second `LAMBDA` is representing the summands as functions over `State` and `Local`. Here, `State` is the set of states and `Local` is the data type $D \times Nat$ of all pairs (d, k) of the summation variables, which is considered as a global variable regarding the property: $p = \sum_{(d,k):local} p$, which is mentioned before.

```
...
State:  TYPE+ = [nat,nat,Buf,Buf,nat,nat,nat,D,nat,nat,nat,D,nat,nat,nat,Buf,Buf,nat]
Local:  TYPE+ = [D,nat]
n, n2:  posnat
e, e2:  D
...
Nmod:lpe[Nnonmod_act, State, Local, 20] =
(# init := (0,0,null,null,0,5,0,e,0,5,0,e,0,0,0,null,null,0),
   sums :=
LAMBDA (i:below(20)) :
LAMBDA (state:State, local:  Local) :
LET (l,m,q,q12,l12,g,h,e,h12,g1,h2,e2,h1,l2,m2,q2,q1,l1) = state,
   (d,k) = local IN
COND
i=0 -> (#
   act := rA(d),
   guard := in_window(l,m,mod(l+n,2*n)),
   next := (l,mod(m+1,2*n),add(d,m,q),q12,l12,g,h,e, h12,g1,h2,e2,h1,l2,m2,q2,q1,l1)
#),
...
i=19 -> (#
   act := sA(retrieve(l12,q12)),
   guard := test(l12,q12),
   next := (l,m,q,remove(l12,q12),mod(l12+1,2*n2),g,h,e, h12,g1,h2,e2,h1,l2,m2,q2,q1,l1)
#)
ENDCOND #)
...
```

Figure 7.3: The formalization of $\mathbf{N}_{mod}$ of SWP with piggybacking in PVS

```
...
l,m,l12,g,h,h12,g1,h2,h1,l2,m2,l1:  var nat
q,q1,q2,q12 :  var Buf
e,e2:  var D
...
inv_6_21_9 (l,m,q,q12,l12,g,h,e,h12,g1,h2,e2,h1,l2,m2,q2,q1,l1):  bool= next_empty(l1,q1)<=m
...
```

Figure 7.4: An example of representing invariants in PVS

7.3 Representing Invariants

By Figure 7.4, we explain how to represent an invariant of the μCRL specification in PVS. Invariants are boolean functions over the set of states. We bring Invariant 6.21.9 in Section 6.6.3 as an example.

7.4 Equality of $\mathbf{N}_{mod}$ and $\mathbf{N}_{nonmod}$

Figure 7.5 is devoted to verify the strong bisimilarity of $\mathbf{N}_{mod}$ and $\mathbf{N}_{nonmod}$ (Proposition 6.22). `state_f` and `local_f` are introduced to construct the state mapping between $\mathbf{N}_{nonmod}$ and $\mathbf{N}_{mod}$.
In PVS we introduce the state mapping (`state_f, local_f`) from the set of states and local variables of $\mathbf{N}_{nonmod}$ to those of $\mathbf{N}_{mod}$. Then we use the corresponding relation to this state mapping, and we show that this relation is a bisimulation relation between $\mathbf{N}_{nonmod}$ and $\mathbf{N}_{mod}$.

We didn't formalize CL-RSP in PVS, because it depends on recursive process equations. This would require a lot of work for embedding μCRL in PVS, which would complicate the formalization too much. In PVS we defined an LPE as a list of summands (not as a recursive equation), equipped with the standard LTS semantics. It could be proved directly that state mappings preserve strong bisimulation. Still, the manual proof is based on CL-RSP, mainly for algebraic reasons: by using algebraic principles only, the stated equivalence still holds in non-standard models for process algebra + CL-RSP.

7.5 Correctness of $\mathbf{N}_{mod}$

Figure 7.6 is devoted to verify the branching bisimilarity of $\mathbf{N}_{mod}$ and $\mathbf{Z}$ (Theorem 6.10). `qlist(q,i,j)` is used to describe the function $q[i..j\rangle$, which is defined as an application on triples. The function `fc(l,m,q,q12,l12,g,h,e,h12,g1,h2,e2,h1, l2,m2,q2,q1,l1)` defines the focus condition for $\mathbf{N}_{nonmod}(\ell, m, q, q_2', \ell_2', g, h, e, h_2',$

```
...
state_f(l,m,q,q12,l12,g,h,e,h12,g1,h2,e2,h1,l2,m2,q2,q1,l1):  State=
   (mod(l,2*n),mod(m,2*n),modulo2(q,2*n),modulo2(q12,2*n2),mod(l12,2*n2),
      g,mod(h,2*n),e,mod(h1,2*n2),g1,mod(h2,2*n2),e2,mod(h1,2*n),
      mod(l2,2*n2),mod(m2,2*n2),modulo2(q2,2*n2),modulo2(q1,2*n),
      mod(l1,2*n)),
local_f(l:Local,i:below(20)):  Local=
   LET (e,k)=l IN
      IF i=4 THEN (e,mod(k,2*n)) ELSE (IF i=9 THEN (e,mod(k,2*n2)) ELSE(e,k)) ENDIF
...
Proposition_6_22:  proposition bisimilar (lpe2lts(Nnonmod),lpe2lts(Nmod))
...
```

Figure 7.5: Equality of $\mathbf{N}_{mod}$ and $\mathbf{N}_{nonmod}$ in PVS

$g', h_2, e_2, h', \ell_2, m_2, q_2, q', \ell'$) as a boolean function on set of states. The state mapping `h` maps states of $\mathbf{N}_{nonmod}$ to states of $\mathbf{Z}$, which is called $\phi : \Xi \to \mathit{List} \times \mathit{List}$ in Section 6.7.2. `k` is a Boolean function which is used to match each external action of $\mathbf{N}_{nonmod}$ to the corresponding one of $\mathbf{Z}$. This is done by corresponding the number of each summand of $\mathbf{N}_{nonmod}$ to one of $\mathbf{Z}$. As PVS requires, this function must be total, therefore without loss of generality we map all the summands with an internal action, from $\mathbf{N}_{nonmod}$'s specification, to the second summand of $\mathbf{Z}$'s specification.

According to cones and foci proof method [FP03], to derive that $\mathbf{N}_{nonmod}$ and $\mathbf{N}_{mod}$ are branching bisimilar, it is enough to check the matching criteria and the reachability of focus points. The two conditions of the cones and foci proof method are represented by `mc` and `WN`, namely matching criteria and the reachability of focus points, respectively. `mc` establishes that all the matching criteria (see Section 6.2) hold for every reachable state `d` in `Nnonmod`, with the aforementioned `h, k` and `fc` functions. `WN` represents the fact that from all reachable states `S` in `Nnonmod`, a focus point can be reached by a finite series of internal actions. The function `lpe2lts` provides the Labeled Transition System semantics of an LPE (see [FPvdP05]).

7.6 Remarks on the Verification in PVS

We used PVS to find the omissions and undetected potential errors that have been ignored in the manual μCRL proofs, some of them have been shown as examples in Section 7.1. PVS guided us to find some important invariants. The PVS verification can be reused to check modifications of the SWP nearly automatically. Also, the generic parts of the PVS formalization can be reused to verify the correctness of many other protocols.
We affirmed the termination of recursive definitions by means of various measure functions. We represented LPEs in PVS and then introduced $\mathbf{N}_{mod}$ and $\mathbf{N}_{nonmod}$ as

```
...
fc(l,m,q,q12,l12,g,h,e,h12,g1,h2,e2,h1,l2,m2,q2,q1,l1):  bool =
      (l=m OR l=l1+n) AND (l2=m2 OR l2=l12+n2)
k(i):  below(2)= IF i=18 THEN 0 ELSE
      IF i=10 THEN 1 ELSE
      IF i=11 THEN 2 ELSE 3 ENDIF ENDIF ENDIF
h(l,m,q,q12,l12,g,h,e,h12,g1,h2,e2,h1,l2,m2,q2,q1,l1):  [List_,List_]=
   (concat(qlist(q1,l1,next_empty(l1,q1)),qlist(q,next_empty(l1,q1),m)),
      concat(qlist(q12,l12,next_empty(l12,q12)),qlist(q2,next_empty(l12,q12),m2)))
mc:  THEOREM FORALL d:  reachable(Nnonmod)(d) IMPLIES MC(Nnonmod,Z,k,h,fc)(d)
WN: LEMMA FORALL S: reachable(Nnonmod)(S) IMPLIES WN(Nnonmod,fc)(S)
main:  THEOREM brbisimilar(lpe2lts(Nmod),lpe2lts(Z))
...
```

Figure 7.6: Correctness of $\mathbf{N}_{mod}$ in PVS

LPEs. We verified the bisimulation of $\mathbf{N}_{nonmod}$ and $\mathbf{N}_{mod}$. Finally we used the cones and foci proof method [FP03], to prove that $\mathbf{N}_{mod}$ and the external behavior of the SWP with piggybacking, represented by $\mathbf{Z}$, are branching bisimilar.

A

Proofs on Properties of Data

This appendix contains proofs of the lemmas in Section 6.6.

A.0.1 Basic Properties

We only prove the last case in Lemma 6.13 on modulo arithmetic,
as the first three cases were already proved in [BFG+05].

Proof.

7. Let $j|_{2n} = k|_{2n}$.

$$
\begin{array}{lll}
 & i \le j \le i+n \ \wedge \ i \le k \le i+n & \\
\rightarrow & j \dot{-} k \le n & \\
\leftrightarrow & ((j\ \mathit{div}\ 2n)\cdot 2n + j|_{2n}) \dot{-} ((k\ \mathit{div}\ 2n)\cdot 2n + k|_{2n}) \le n & \text{(Lem. 6.13.3)} \\
\leftrightarrow & (j\ \mathit{div}\ 2n)\cdot 2n \dot{-} (k\ \mathit{div}\ 2n)\cdot 2n \le n & (j|_{2n} = k|_{2n}) \\
\leftrightarrow & (j\ \mathit{div}\ 2n)\cdot 2 \dot{-} (k\ \mathit{div}\ 2n)\cdot 2 \le 1 & \\
\rightarrow & (j\ \mathit{div}\ 2n) \ge (k\ \mathit{div}\ 2n) &
\end{array}
$$

By symmetry, also $(j\ \mathit{div}\ 2n) \le (k\ \mathit{div}\ 2n)$, so
$(j\ \mathit{div}\ 2n) = (k\ \mathit{div}\ 2n)$. Since $j|_{2n} = k|_{2n}$,
by Lem. 6.13.3, $j = k$.

■

We only prove the last case in Lemma 6.14 on basic properties of buffers,
as the first six cases were already proved in [BFG+05].

Proof.

7. If $i = j$ then the lemma is trivial. Let $i \neq j$. We use induction on the structure of q.

   - $q = []$.
     Trivial.
   - $q = inb(d', k, q')$.
     * $j = k$.

$$\begin{aligned}
& remove(i, remove(j, inb(d', k, q'))) & \\
&= remove(i, remove(j, q')) & (j = k)\\
&= remove(j, remove(i, q')) & \text{(i.h.)}\\
&= remove(j, inb(d', k, remove(i, q'))) & (j = k)\\
&= remove(j, remove(i, inb(d', k, q'))) & (i \neq k)
\end{aligned}$$

     * $j \neq k$.
       · $i = k$.

$$\begin{aligned}
& remove(i, remove(j, inb(d', k, q'))) & \\
&= remove(i, inb(d', k, remove(j, q'))) & (j \neq k)\\
&= remove(i, remove(j, q')) & (i = k)\\
&= remove(j, remove(i, q')) & \text{(i.h.)}\\
&= remove(j, remove(i, inb(d', k, q'))) & (i = k)
\end{aligned}$$

       · $i \neq k$.

$$\begin{aligned}
& remove(i, remove(j, inb(d', k, q'))) & \\
&= remove(i, inb(d', k, remove(j, q'))) & (j \neq k)\\
&= inb(d', k, remove(i, remove(j, q'))) & (i \neq k)\\
&= inb(d', k, remove(j, remove(i, q'))) & \text{(i.h.)}\\
&= remove(j, inb(d', k, remove(i, q'))) & (j \neq k)\\
&= remove(j, remove(i, inb(d', k, q'))) & (i \neq k)
\end{aligned}$$

■

The cases of Lemma 6.15 on the *next-empty* function were all proved in [BFG+05]. This brings us to Lemma 6.16, on modulo arithmetic for buffers. Only cases 1 and 4 were not yet proved in [BFG+05], because of our new definition of $\textit{next-empty}|_{2n}(i,q)$. So we prove these two cases here.

Proof.

1. We have $i|_{2n} < 2n$ (Lem. 6.13.2).

   - $i|_{2n} = 2n \dot{-} 1$.
     * $\neg \textit{test}(i|_{2n}, q)$.

       Hence $\textit{next-empty}(i|_{2n}, q) = i|_{2n} < 2n$. Then

       $$\begin{aligned}&\textit{next-empty}|_{2n}(i,q)\\ &= \textit{next-empty}(i|_{2n}, q) \quad (\textit{next-empty}(i|_{2n}, q) < 2n)\\ &= i|_{2n}\end{aligned}$$

     * $\textit{test}(i|_{2n}, q)$.

       Hence $\textit{next-empty}(i|_{2n}, q) = \textit{next-empty}(S(i|_{2n}), q) = \textit{next-empty}(2n, q) \geq 2n$ (Lem. 6.15.2).
       Then

       $$\begin{aligned}&\textit{next-empty}|_{2n}(i,q)\\ &= \textit{if}\,(\textit{next-empty}(0,q) < 2n, \textit{next-empty}(0,q),\ 2n)\\ &= \textit{next-empty}|_{2n}(0,q)\\ &= \textit{next-empty}|_{2n}((2n)|_{2n}, q)\\ &= \textit{next-empty}|_{2n}(S(i|_{2n})|_{2n}, q) && (i|_{2n} = 2n \dot{-} 1)\\ &= \textit{next-empty}|_{2n}(S(i)|_{2n}, q) && (\text{Lem. } 6.13.1)\end{aligned}$$

   - $i|_{2n} < 2n \dot{-} 1$.

     * $\neg \textit{test}(i|_{2n}, q)$. Hence

       $\textit{next-empty}(i|_{2n}, q) = i|_{2n} < 2n \dot{-} 1$. Then

       $$\begin{aligned}&\textit{next-empty}|_{2n}(i,q)\\ &= \textit{next-empty}(i|_{2n}, q) && (\textit{next-empty}(i|_{2n}, q) < 2n \dot{-} 1)\\ &= i|_{2n} && (\neg \textit{test}(i|_{2n}, q))\end{aligned}$$

* $test(i|_{2n}, q)$.

  We recall the assumption $S(i|_{2n}) < 2n$.
  Therefore $S(i|_{2n}) = S(i|_{2n})|_{2n} = S(i)|_{2n}$
  (Lem. 6.13.1). Using this and the assumption
  $test(i|_{2n}, q)$, we get

  $next\text{-}empty(i|_{2n}, q) = next\text{-}empty(S(i|_{2n}), q) = next\text{-}empty(S(i)|_{2n}, q)$.
  Then

$$
\begin{aligned}
& next\text{-}empty|_{2n}(i, q) \\
= {} & if\,(next\text{-}empty(i|_{2n}, q) < 2n,\; next\text{-}empty(i|_{2n}, q), \\
& \quad if\,(next\text{-}empty(0, q) < 2n, next\text{-}empty(0, q),\; 2n)) \\
= {} & if\,(next\text{-}empty(S(i)|_{2n}, q) < 2n,\; next\text{-}empty(S(i)|_{2n}, q), \\
& \quad if\,(next\text{-}empty(0, q) < 2n, next\text{-}empty(0, q),\; 2n)) \\
= {} & if\,(next\text{-}empty((S(i)|_{2n})|_{2n}, q) < 2n,\; next\text{-}empty((S(i)|_{2n})|_{2n}, q), \\
& \quad if\,(next\text{-}empty(0, q) < 2n, next\text{-}empty(0, q),\; 2n)) \qquad \text{(Lem. 6.13.1)} \\
= {} & next\text{-}empty|_{2n}(S(i)|_{2n}, q)
\end{aligned}
$$

4. By induction on $(i + n) \dot{-} k$. Let $test(j, q) \to i \leq j < i + n$.

   - $k = i + n$. Then $\neg test(k, q)$, since $test(j, q) \to i \leq j < i + n$.
     So by Lemma 6.16.2,
     $\neg test(k|_{2n}, q|_{2n})$, and hence
     by Lemma 6.13.1,
     $\neg test((k|_{2n})|_{2n}, q|_{2n})$.
     Hence,

$$
\begin{aligned}
& next\text{-}empty|_{2n}(k|_{2n}, q|_{2n}) \\
= {} & (k|_{2n})|_{2n} && (\neg test((k|_{2n})|_{2n}, q|_{2n}),\; \text{Lem. 6.16.1}) \\
= {} & k|_{2n} && \text{(Lem. 6.13.1)} \\
= {} & next\text{-}empty(k, q)|_{2n} && (\neg test(k, q))
\end{aligned}
$$

   - $i \leq k < i + n$. Then $i \leq S(k) \leq i + n$.

     * $\neg test(k, q)$. Similarly.
     * $test(k, q)$.
       By Lemma 6.16.2, also
       $test(k|_{2n}, q|_{2n})$. Hence

$test((k|_{2n})|_{2n}, q|_{2n})$
by Lemma 6.13.1.
Hence,

$$\begin{array}{ll}
 & next\text{-}empty|_{2n}(k|_{2n}, q|_{2n}) \\
= & next\text{-}empty|_{2n}(S(k|_{2n})|_{2n}, q|_{2n}) \quad (test((k|_{2n})|_{2n}, q|_{2n}),\ \text{Lem. 6.16.1}) \\
= & next\text{-}empty|_{2n}(S(k)|_{2n}, q|_{2n}) \quad (\text{Lem. 6.13.1}) \\
= & next\text{-}empty(S(k), q)|_{2n} \quad (\text{i.h.}) \\
= & next\text{-}empty(k, q)|_{2n} \quad (test(k, q))
\end{array}$$

■

Only the last two cases of Lemma 6.17 on lists were not yet proved in [BFG+05].

Proof.

8. By induction on $j \dot{-} i$.

   - $j \dot{-} i = 0$. Then $j = i$, since by assumption $i \leq j$.

$$\begin{array}{lll}
 & append(d, q[i..j\rangle) & \\
= & append(d, \langle\rangle) & (j = i) \\
= & inl(d, \langle\rangle) & \\
= & inl(retrieve(i, add(d, i, q)), add(d, i, q)[S(i)..S(i)\rangle) & (\text{Lem. 6.18.4}) \\
= & add(d, i, q)[i..S(i)\rangle & \\
= & add(d, j, q)[i..S(j)\rangle & (j = i)
\end{array}$$

   - $j > i$.

$$\begin{array}{lll}
 & append(d, q[i..j\rangle) & \\
= & append(d, inl(retrieve(i, q), q[S(i)..j\rangle)) & (j > i) \\
= & inl(retrieve(i, q), append(d, q[S(i)..j\rangle)) & \\
= & inl(retrieve(i, q), add(d, j, q)[S(i)..S(j)\rangle) & (\text{i.h.}) \\
= & inl(retrieve(i, add(d, j, q)), add(d, j, q)[S(i)..S(j)\rangle) & (j > i, \text{Lem. 6.18.4}) \\
= & add(d, j, q)[i..S(j)\rangle & (S(j) > i))
\end{array}$$

9. By induction on $j \dot{-} i$.

   - $j \dot{-} i = 0$. So $j \leq i$. Then by definition both sides are $\langle\rangle$.
   - $j > i$.

By Lemma 6.18.4,

$retrieve(i, add(retrieve(k, q), k, q)) = if(i = k, retrieve(k, q), retrieve(i, q)) = retrieve(i, q)$

$$\begin{array}{lll} = & add(retrieve(k,q),k,q)[i..j\rangle & \\ = & inl(retrieve(i, add(retrieve(k,q),k,q)), add(retrieve(k,q),k,q)[S(i)..j\rangle) & (j > i) \\ = & inl(retrieve(i,q), add(retrieve(k,q),k,q)[S(i)..j\rangle) & \text{(above)} \\ = & inl(retrieve(i,q), q[S(i)..j\rangle) & \text{(i.h.)} \\ = & q[i..j\rangle & (j > i) \end{array}$$

■

A.0.2 Ordered Buffers

We proceed to prove Lemma 6.18 on the *add* function in its entirety.

Proof.

1. By induction on the structure of q.

   - $q = []$.
     Trivial.
   - $q = inb(d', k, q')$.
     - $j > k$.

$$\begin{array}{ll} & test(i,q) \\ = & test(i, inb(d',k,q')) \\ = & (i = k) \vee test(i,q') \\ \rightarrow & (i = k) \vee test(i, add(d,j,q')) \quad \text{(i.h.)} \\ = & test(i, inb(d',k, add(d,j,q'))) \\ = & test(i, add(d,j, inb(d',k,q'))) \quad (j > k) \\ = & test(i, add(d,j,q)) \end{array}$$

     - $j \le k$.

$$
\begin{aligned}
& test(i,q) \\
\rightarrow\ & (i=j) \vee test(i,q) \\
=\ & (i=j) \vee (test(i,q) \wedge i \neq j) \\
=\ & (i=j) \vee test(i, remove(j,q)) && \text{(Lem. 6.14.3)} \\
=\ & test(i, inb(d,j,remove(j,q))) \\
=\ & test(i, add(d,j,q)) && (j \leq k,\ q = inb(d',k,q'))
\end{aligned}
$$

2. By induction on $S(max(q)) \dot{-} i$.

   - $S(max(q)) \dot{-} i = 0$.
     Therefore $\neg test(i,q)$ by
     Lemma 6.14.1.

$$
\begin{aligned}
& next\text{-}empty(i, add(d,j,q)) \\
\geq\ & i && \text{(Lem. 6.15.2)} \\
=\ & next\text{-}empty(i,q)
\end{aligned}
$$

   - $S(max(q)) \dot{-} i > 0$.
     - $\neg test(i,q)$. Similarly.
     - $test(i,q)$. Hence $test(i, add(d,j,q))$ by
       Lemma 6.18.1.

$$
\begin{aligned}
& next\text{-}empty(i, add(d,j,q)) \\
=\ & next\text{-}empty(S(i), add(d,j,q)) \\
\geq\ & next\text{-}empty(S(i), q) && \text{(i.h.)} \\
=\ & next\text{-}empty(i,q)
\end{aligned}
$$

3. By induction on the structure of q.

   - $q = []$.
     Trivial.
   - $q = inb(d',k,q')$.

– $j > k$.

$$\begin{aligned}
& \mathit{test}(i, \mathit{add}(d, j, q)) \\
=\ & \mathit{test}(i, \mathit{add}(d, j, \mathit{inb}(d', k, q'))) \\
=\ & \mathit{test}(i, \mathit{inb}(d', k, \mathit{add}(d, j, q'))) \\
=\ & (i = k) \vee \mathit{test}(i, \mathit{add}(d, j, q')) \\
=\ & (i = k) \vee (i = j) \vee \mathit{test}(i, q') && \text{(i.h.)} \\
=\ & (i = j) \vee \mathit{test}(i, \mathit{inb}(d', k, q')) \\
=\ & (i = j) \vee \mathit{test}(i, q)
\end{aligned}$$

– $j \leq k$. Since $q = \mathit{inb}(d', k, q')$,

$$\begin{aligned}
& \mathit{test}(i, \mathit{add}(d, j, q)) \\
=\ & \mathit{test}(i, \mathit{inb}(d, j, \mathit{remove}(j, q))) \\
=\ & (i = j) \vee \mathit{test}(i, \mathit{remove}(j, q)) \\
=\ & (i = j) \vee (\mathit{test}(i, q) \wedge i \neq j) && \text{(Lem. 6.14.3)} \\
=\ & (i = j) \vee \mathit{test}(i, q)
\end{aligned}$$

4. By induction on the structure of q.

- $q = []$.
  Trivial.
- $q = \mathit{inb}(d', k, q')$.
  – $j > k$.

$$\begin{aligned}
& \mathit{retrieve}(i, \mathit{add}(d, j, \mathit{inb}(d', k, q'))) \\
=\ & \mathit{retrieve}(i, \mathit{inb}(d', k, \mathit{add}(d, j, q'))) \\
=\ & \mathit{if}\,(i = k, d', \mathit{retrieve}(i, \mathit{add}(d, j, q'))) \\
=\ & \mathit{if}\,(i = k, d', \mathit{if}\,(i{=}j, d, \mathit{retrieve}(i, q'))) && \text{(i.h.)} \\
=\ & \mathit{if}\,(i = j, d, \mathit{if}\,(i{=}k, d', \mathit{retrieve}(i, q'))) && (j > k) \\
=\ & \mathit{if}\,(i = j, d, \mathit{retrieve}(i, q))
\end{aligned}$$

  – $j \leq k$.

$$\begin{aligned}
&retrieve(i, add(d, j, q)) \\
&= retrieve(i, inb(d, j, remove(j, q))) \\
&= if(i = j, d, retrieve(i, remove(j, q))) \\
&= if(i = j, d, retrieve(i, q)) && \text{(Lem. 6.14.4)}
\end{aligned}$$

5. By induction on the structure of q.

   - $q = []$.
     Trivial.
   - $q = inb(d', k, q')$.

     Using Lemma 6.14.3 we get
     $test(i, remove(i, q)) = (test(i, q) \wedge i \neq i)$. Hence
     $\neg test(i, remove(i, q))$. Using this with
     Lemma 6.14.2 we derive
     $remove(i, remove(i, q)) = remove(i, q)$.

     – $i > k$.

$$\begin{aligned}
&remove(i, add(d, i, inb(d', k, q'))) \\
&= remove(i, inb(d', k, add(d, i, q'))) && (i > k) \\
&= inb(d', k, remove(i, add(d, i, q'))) && (i \neq k) \\
&= inb(d', k, remove(i, q')) && \text{(i.h.)} \\
&= remove(i, inb(d', k, q')) && (i \neq k)
\end{aligned}$$

     – $i \leq k$. Since $q = inb(d', k, q')$,

$$\begin{aligned}
&remove(i, add(d, i, q)) \\
&= remove(i, inb(d, i, remove(i, q))) && (i \leq k) \\
&= remove(i, remove(i, q)) \\
&= remove(i, q) && \text{(above)}
\end{aligned}$$

6. We assume $j \neq next\text{-}empty(i, q)$. Then

we prove the lemma using induction on $S(max(q)) \dot{-} i$.

- $S(max(q)) \dot{-} i = 0$.
  Therefore $\neg test(i, q)$ by
  Lemma 6.14.1, so that
  $next\text{-}empty(i, q) = i$. An immediate consequence of this is
  that $j \neq i$ with respect to the assumption. Hence
  $\neg test(i, add(d, j, q))$ because of
  Lemma 6.18.3. Then

$$\begin{aligned} & next\text{-}empty(i, add(d, j, q)) \\ = {} & i \\ = {} & next\text{-}empty(i, q) \end{aligned}$$

- $S(max(q)) \dot{-} i > 0$.
  - $\neg test(i, q)$. Similarly.
  - $test(i, q)$. Hence $test(i, add(d, j, q))$ by
    Lemma 6.18.1. Also
    $next\text{-}empty(i, q) = next\text{-}empty(S(i), q)$ and so
    $next\text{-}empty(S(i), q) \neq j$.

$$\begin{aligned} & next\text{-}empty(i, add(d, j, q)) & \\ = {} & next\text{-}empty(S(i), add(d, j, q)) & (test(i, add(d, j, q))) \\ = {} & next\text{-}empty(S(i), q) & \text{(i.h.)} \\ = {} & next\text{-}empty(i, q) & (test(i, q)) \end{aligned}$$

7. By induction on $S(max(q)) \dot{-} i$.

- $S(max(q)) \dot{-} i = 0$.
  Therefore $\neg test(i, q)$ by
  Lemma 6.14.1.
  So
  $next\text{-}empty(i, q) = i$. Moreover, using Lemma 6.18.3
  we get $test(i, add(d, i, q))$. Then

$$\begin{aligned}
& \mathit{next\text{-}empty}(i, \mathit{add}(d, \mathit{next\text{-}empty}(i,q), q)) \\
&= \mathit{next\text{-}empty}(i, \mathit{add}(d, i, q)) \\
&= \mathit{next\text{-}empty}(S(i), \mathit{add}(d, i, q)) && (\mathit{test}(i, \mathit{add}(d, j, q))) \\
&= \mathit{next\text{-}empty}(S(i), q) && \text{(Lem. 6.18.6)} \\
&= \mathit{next\text{-}empty}(S(\mathit{next\text{-}empty}(i,q)), q)
\end{aligned}$$

- $S(\mathit{max}(q)) \dot{-} i > 0$.
  - $\neg \mathit{test}(i,q)$. Similarly.
  - $\mathit{test}(i,q)$. Hence $\mathit{test}(i, \mathit{add}(d, \mathit{next\text{-}empty}(i,q), q))$ by Lemma 6.18.1. Then

$$\begin{aligned}
& \mathit{next\text{-}empty}(i, \mathit{add}(d, \mathit{next\text{-}empty}(i,q), q)) \\
&= \mathit{next\text{-}empty}(S(i), \mathit{add}(d, \mathit{next\text{-}empty}(i,q), q)) \\
&= \mathit{next\text{-}empty}(S(\mathit{next\text{-}empty}(S(i),q)), q) && \text{(i.h.)} \\
&= \mathit{next\text{-}empty}(S(\mathit{next\text{-}empty}(i,q)), q) && (\mathit{test}(i,q))
\end{aligned}$$

8. By induction on the structure of q.

- $q = []$.
  Trivial.
- $q = \mathit{inb}(d', k, q')$.
  - $i = k$.

$$\begin{aligned}
& \mathit{remove}(i, \mathit{add}(d, j, q)) \\
&= \mathit{remove}(i, \mathit{inb}(d', i, \mathit{add}(d, j, q'))) && (i < j) \\
&= \mathit{remove}(i, \mathit{add}(d, j, q')) \\
&= \mathit{add}(d, j, \mathit{remove}(i, q')) && \text{(i.h.)} \\
&= \mathit{add}(d, j, \mathit{remove}(i, \mathit{inb}(d', i, q'))) \\
&= \mathit{add}(d, j, \mathit{remove}(i, q))
\end{aligned}$$

  - $i \neq k$.
    * $j \leq k$. So $i < j \leq k$. Hence

$$\begin{aligned}
& remove(i, add(d, j, q)) \\
=\ & remove(i, inb(d, j, remove(j, q))) && (j \leq k) \\
=\ & inb(d, j, remove(i, remove(j, q))) && (i < j) \\
=\ & inb(d, j, remove(j, remove(i, q))) && (\text{Lem. } 6.14.7) \\
=\ & inb(d, j, remove(j, inb(d', k, remove(i, q')))) && (i \neq k) \\
=\ & add(d, j, inb(d', k, remove(i, q'))) && (j \leq k) \\
=\ & add(d, j, remove(i, q)) && (i \neq k)
\end{aligned}$$

* $j > k$.

$$\begin{aligned}
& remove(i, add(d, j, q)) \\
=\ & remove(i, inb(d', k, add(d, j, q'))) && (j > k) \\
=\ & inb(d', k, remove(i, add(d, j, q'))) && (i \neq k) \\
=\ & inb(d', k, add(d, j, remove(i, q'))) && (\text{i.h.}) \\
=\ & add(d, j, inb(d', k, remove(i, q'))) && (j > k) \\
=\ & add(d, j, remove(i, q)) && (i \neq k)
\end{aligned}$$

9. We only prove the lemma for $i < j$, by symmetry it then holds for the other case too. We use induction on the structure of q.

   - $q = []$.
     Trivial.
   - $q = inb(d', k, q')$,
     - $j \leq k$.

$$\begin{aligned}
& add(e, i, add(d, j, q)) \\
=\ & inb(e, i, remove(i, add(d, j, q))) && (i < j) \\
=\ & inb(e, i, add(d, j, remove(i, q))) && (i < j, \text{ Lem. } 6.18.8) \\
=\ & add(d, j, inb(e, i, remove(i, q))) && (i < j) \\
=\ & add(d, j, add(e, i, q)) && (i < j \leq k)
\end{aligned}$$

     - $j > k$.

* $i \le k$.

$$\begin{aligned} & add(e,i,add(d,j,q)) \\ = {} & add(e,i,inb(d',k,add(d,j,q')) && (j > k) \\ = {} & inb(e,i,remove(i,inb(d',k,add(d,j,q')))) && (i \le k) \\ = {} & inb(e,i,remove(i,add(d,j,q))) && (j > k) \\ = {} & inb(e,i,add(d,j,remove(i,q))) && (\text{Lem. 6.18.8}, i < j) \\ = {} & add(d,j,inb(e,i,remove(i,q))) && (j > i) \\ = {} & add(d,j,add(e,i,q)) && (i \le k) \end{aligned}$$

* $i > k$.

$$\begin{aligned} & add(e,i,add(d,j,q)) \\ = {} & add(e,i,inb(d',k,add(d,j,q'))) && (j > i > k) \\ = {} & inb(d',k,add(e,i,add(d,j,q'))) && (i > k) \\ = {} & inb(d',k,add(d,j,add(e,i,q'))) && (\text{i.h.}) \\ = {} & add(d,j,inb(d',k,add(e,i,q'))) && (j > i > k) \\ = {} & add(d,j,add(e,i,q)) && (i > k) \end{aligned}$$

■

Now we prove Lemma 6.19 on the functions *smaller* and *sort* in its entirety.

Proof.

1. By induction on the structure of q.

   - $q = []$.
     Trivial.
   - $q = inb(d',k,q')$.
     - $j = k$.

$$\begin{aligned} & smaller(i,inb(d',k,q')) \\ = {} & i < k \wedge smaller(i,q') \\ \rightarrow {} & smaller(i,remove(j,q')) && (\text{i.h.}) \\ = {} & smaller(i,remove(j,inb(d',k,q'))) && (j = k) \end{aligned}$$

– $j \neq k$.

$$\begin{array}{ll} & smaller(i, inb(d', k, q')) \\ = & i < k \wedge smaller(i, q') \\ \rightarrow & i < k \wedge smaller(i, remove(j, q')) \quad \text{(i.h.)} \\ = & smaller(i, inb(d', k, remove(j, q'))) \\ = & smaller(i, remove(j, inb(d', k, q'))) \quad (j \neq k) \end{array}$$

2. By induction on the structure of q.

   - $q = []$.

     Trivial.

   - $q = inb(d', k, q')$.

     – $j \leq k$.

$$\begin{array}{ll} & i < j \wedge smaller(i, q) \\ \rightarrow & i < j \wedge smaller(i, remove(j, q)) \quad \text{(Lem. 6.19.1)} \\ = & smaller(i, inb(d, j, remove(j, q))) \\ = & smaller(i, add(d, j, q)) \quad (j \leq k) \end{array}$$

     – $j > k$.

$$\begin{array}{ll} & i < j \wedge smaller(i, inb(d', k, q')) \\ = & i < j \wedge i < k \wedge smaller(i, q') \\ \rightarrow & i < k \wedge smaller(i, add(d, j, q')) \quad \text{(i.h.)} \\ = & smaller(i, inb(d', k, add(d, j, q'))) \\ = & smaller(i, add(d, j, inb(d', k, q'))) \quad (j > k) \end{array}$$

3. By induction on the structure of q.

   - $q = []$.

     Trivial.

   - $q = inb(d', k, q')$.

$$\begin{array}{ll} & smaller(i, inb(d', k, q')) \\ = & i < k \wedge smaller(i, q') \\ \to & i < k \wedge (remove(i, q') = q') \qquad \text{(i.h.)} \\ \to & i < k \wedge (inb(d', k, remove(i, q')) = inb(d', k, q')) \\ = & i < k \wedge (remove(i, inb(d', k, q')) = inb(d', k, q')) \\ \to & remove(i, inb(d', k, q')) = inb(d', k, q') \end{array}$$

4. By induction on the structure of q.

- $q = []$.

  Trivial.

- $q = inb(d', k, q')$.

$$\begin{array}{ll} & i < j \wedge smaller(j, inb(d', k, q')) \\ = & i < j \wedge j < k \wedge smaller(j, q') \\ \to & i < k \wedge smaller(i, q') \qquad \text{(i.h.)} \\ = & smaller(i, inb(d', k, q')) \end{array}$$

5. By induction on the structure of q.

- $q = []$.

  Trivial.

- $q = inb(d', k, q')$.

  - $i < k$. $sort(q)$, so $smaller(k, q')$. Since $i < k$, by Lemma 6.19.4 $smaller(i, q')$, and so $smaller(i, q)$. Then $remove(i, q) = q$ by Lemma 6.19.3.

$$\begin{array}{lll} & sort(q) & \\ = & sort(q) \wedge smaller(i, q) & (smaller(i, q)) \\ = & sort(remove(i, q)) \wedge smaller(i, remove(i, q)) & \text{(above)} \\ = & sort(inb(d, i, remove(i, q))) & \\ = & sort(add(d, i, q)) & (i < k) \end{array}$$

  - $i = k$.

$$\begin{array}{lll} & sort(inb(d',k,q')) & \\ = & smaller(i,q') \wedge sort(q') & (i=k) \\ \to & smaller(i,remove(i,q')) \wedge sort(remove(i,q')) & \text{(Lem. 6.19.3, Lem. 6.19.1)} \\ = & sort(inb(d,i,remove(i,q'))) & \\ = & sort(inb(d,i,remove(i,inb(d',k,q')))) & (i=k) \\ = & sort(add(d,i,inb(d',k,q')) & (i=k) \end{array}$$

- $i > k$.

$$\begin{array}{lll} & sort(inb(d',k,q')) & \\ = & smaller(k,q') \wedge sort(q') & \\ \to & smaller(k,add(d,i,q')) \wedge sort(add(d,i,q')) & (i>k, \text{Lem. 6.19.2, i.h.}) \\ = & sort(inb(d',k,add(d,i,q'))) & \\ = & sort(add(d,i,inb(d',k,q'))) & (i>k) \end{array}$$

6. By induction on the structure of q.
   - $q = []$.
     Trivial.
   - $q = inb(d',k,q')$. Then $smaller(i,q)$ implies $i < k$, so

$$\begin{array}{lll} & add(d,i,q) & \\ = & inb(d,i,remove(i,q)) & (i<k) \\ = & inb(d,i,q) & (smaller(i,q), \text{Lem. 6.19.3}) \end{array}$$

7. By induction on the structure of q.
   - $q = []$.
     Trivial.
   - $q = inb(d',k,q')$. Let $sort(q) \wedge j < i$.
     - $i = k$.

$$\begin{array}{lll} & sort(inb(d',k,q')) & \\ \to & smaller(i,q') & (i=k) \\ \to & smaller(j,q') & (j<i, \text{Lem. 6.19.4}) \\ \leftrightarrow & smaller(j,q) & (j<i=k) \\ \to & remove(j,q)=q & \text{(Lem. 6.19.3)} \end{array}$$

Hence

$$\begin{array}{lll} & remove(i, add(d,j,q)) & \\ = & remove(i, inb(d,j,remove(j,q))) & (j < k) \\ = & remove(i, inb(d,j,q)) & \text{(above)} \\ = & inb(d,j,remove(i,q)) & (j < i) \\ = & inb(d,j,remove(i,q')) & (i = k) \\ = & remove(i, inb(d,j,q')) & (j < i,\ i = k) \\ = & remove(i, add(d,j,q')) & (smaller(j,q'), \text{ Lem. 6.19.6}) \\ = & add(d,j,remove(i,q')) & \text{(i.h.)} \\ = & add(d,j,remove(i,q)) & (i = k) \end{array}$$

– $i < k$.

$$\begin{array}{lll} & sort(inb(d',k,q')) & \\ \to & smaller(k,q') & \\ \to & smaller(i,q') & (i < k, \text{ Lem. 6.19.4}) \\ \leftrightarrow & smaller(i,q) & (i < k) \\ \to & smaller(j,q) & (j < i, \text{ Lem. 6.19.4}) \\ \to & remove(j,q) = q & \text{(Lem. 6.19.3)} \end{array}$$

Hence

$$\begin{array}{lll} & remove(i, add(d,j,q)) & \\ = & remove(i, inb(d,j,remove(j,q))) & (j < i < k) \\ = & remove(i, inb(d,j,q)) & \text{(above)} \\ = & inb(d,j,remove(i,q)) & (j < i) \\ = & add(d,j,remove(i,q)) & (smaller(j,q), \text{ Lem. 6.19.1, Lem. 6.19.6}) \end{array}$$

– $i > k$.

∗ $k = j$.

$$
\begin{array}{lll}
& remove(i, add(d, j, q)) & \\
= & remove(i, inb(d, j, remove(j, q))) & (j = k) \\
= & remove(i, inb(d, j, remove(j, q'))) & (j = k) \\
= & inb(d, j, remove(i, remove(j, q'))) & (j = k < i) \\
= & inb(d, j, remove(j, remove(i, q'))) & \text{(Lem. 6.14.7)} \\
= & inb(d, j, remove(j, inb(d', j, remove(i, q')))) & \\
= & add(d, j, inb(d', j, remove(i, q'))) & \\
= & add(d, j, remove(i, inb(d', j, q'))) & (j < i) \\
= & add(d, j, remove(i, q)) & (j = k)
\end{array}
$$

* $k > j$.

$$
\begin{array}{lll}
& remove(i, add(d, j, q)) & \\
= & remove(i, inb(d, j, remove(j, q))) & (j < k) \\
= & inb(d, j, remove(i, remove(j, q))) & (j < i) \\
= & inb(d, j, remove(j, remove(i, q))) & \text{(Lem. 6.14.7)} \\
= & inb(d, j, remove(j, inb(d', k, remove(i, q')))) & \\
= & add(d, j, inb(d', k, remove(i, q'))) & (j < k) \\
= & add(d, j, remove(i, q)) & (k < i)
\end{array}
$$

* $k < j$.

$$
\begin{array}{lll}
& remove(i, add(d, j, q)) & \\
= & remove(i, inb(d', k, add(d, j, q'))) & (k < j) \\
= & inb(d', k, remove(i, add(d, j, q'))) & (k < j < i) \\
= & inb(d', k, add(d, j, remove(i, q'))) & (\text{i.h., } sort(q'),\ j < i) \\
= & add(d, j, inb(d', k, remove(i, q'))) & (k < j) \\
= & add(d, j, remove(i, q)) & (k < j < i)
\end{array}
$$

8. By induction on the structure of q.

- $q = []$.

  Trivial.

- $q = inb(d', k, q')$.

  - $i < k$.

$$
\begin{aligned}
& smaller(k, q') \\
\rightarrow\ & smaller(i, q') && \text{(Lem. 6.19.4)} \\
\rightarrow\ & remove(i, q') = q' && \text{(Lem. 6.19.3)}
\end{aligned}
$$

Hence

$$
\begin{aligned}
& add(d, i, remove(i, q)) \\
=\ & add(d, i, inb(d', k, remove(i, q'))) && (i < k) \\
=\ & add(d, i, q) && \text{(above)}
\end{aligned}
$$

– $i = k$.

$$
\begin{aligned}
& add(d, i, remove(i, q)) \\
=\ & add(d, k, remove(k, q')) && (i = k) \\
=\ & add(d, k, q') && \text{(i.h.)} \\
=\ & inb(d, k, q') && (smaller(k, q'), \text{ Lem. 6.19.6}) \\
=\ & inb(d, k, remove(k, q')) && (smaller(k, q'), \text{ Lem. 6.19.3}) \\
=\ & inb(d, k, remove(k, q)) \\
=\ & add(d, i, q) && (i = k)
\end{aligned}
$$

– $i > k$.

$$
\begin{aligned}
& add(d, i, remove(i, q)) \\
=\ & add(d, i, inb(d', k, remove(i, q'))) && (i > k) \\
=\ & inb(d', k, add(d, i, remove(i, q'))) && (i > k) \\
=\ & inb(d', k, add(d, i, q')) && \text{(i.h.)} \\
=\ & add(d, i, q) && (i > k)
\end{aligned}
$$

■

Now we prove Lemma 6.20 on $q\|_n$ in its entirety.

Proof.

1. By induction on q.

- $q = []$.
  Trivial.
- $q = inb(d, k, q')$.

$$\begin{aligned} & sort(inb(d,k,q')\|_n) \\ = & sort(add(d, k|_n, q'\|_n)) \\ = & \texttt{true} && \text{(i.h., Lem. 6.19.5)} \end{aligned}$$

2. By induction on q.

- $q = []$.
  Trivial.
- $q = inb(d, k, q')$.

$$\begin{aligned} & test(i, q|_n) \\ = & test(i, inb(d, k|_n, q'|_n)) \\ = & (i = k|_n \vee test(i, q'|_n)) \\ = & (i = k|_n \vee test(i, q'\|_n) && \text{(i.h.)} \\ = & test(i, add(d, k|_n, q'\|_n)) && \text{(Lem. 6.18.3)} \\ = & test(i, q\|_n) \end{aligned}$$

3. By induction on q.

- $q = []$.
  Trivial.
- $q = inb(d, k, q')$.

$$\begin{aligned} & retrieve(i, q|_n) \\ = & retrieve(i, inb(d, k|_n, q'|_n)) \\ = & if(i = k|_n, d, retrieve(i, q'|_n)) \\ = & if(i = k|_n, d, retrieve(i, q'\|_n) && \text{(i.h.)} \\ = & retrieve(i, add(d, k|_n, q'\|_n)) && \text{(Lem. 6.18.4)} \\ = & retrieve(i, q\|_n) \end{aligned}$$

4. - $j < i$.

     It holds by Lemma 6.19.7 and Lemma 6.20.1.
   - $i < j$.

     It holds by Lemma 6.18.8.
5. By induction on $(i + n) \dot{-} k$.
   - $k = i + n$.

     $\neg test(k, q)$, since $test(j, q) \to i \leq j < i + n$ by the assumption. So by Lemma 6.16.2, $\neg test(k|_{2n}, q|_{2n})$, and hence by Lemma 6.20.2, $\neg test(k|_{2n}, q\|_{2n})$. Hence

$$\begin{aligned}
& \textit{next-empty}|_{2n}(k|_{2n}, q|_{2n}) \\
&= (k|_{2n})|_{2n} && (\neg test((k|_{2n})|_{2n}, q|_{2n}),\ \text{Lem. 6.16.1}) \\
&= \textit{next-empty}|_{2n}(k|_{2n}, q\|_{2n}) && (\neg test((k|_{2n})|_{2n}, q\|_{2n}),\ \text{Lem. 6.16.1})
\end{aligned}$$

   - $i \leq k < i + n$.
     - $\neg test(k, q)$. Similarly.
     - $test(k, q)$.
       By Lemma 6.16.2, $test(k|_{2n}, q|_{2n})$. Hence $test(k|_{2n}, q\|_{2n})$ by Lemma 6.20.2. Therefore,

$$\begin{aligned}
& \textit{next-empty}|_{2n}(k|_{2n}, q|_{2n}) \\
&= \textit{next-empty}|_{2n}(S(k|_{2n})|_{2n}, q|_{2n}) && (test((k|_{2n})|_{2n}, q|_{2n}),\ \text{Lem. 6.16.1}) \\
&= \textit{next-empty}|_{2n}(S(k)|_{2n}, q|_{2n}) && (\text{Lem. 6.13.1}) \\
&= \textit{next-empty}|_{2n}(S(k)|_{2n}, q\|_{2n}) && (\text{i.h.}) \\
&= \textit{next-empty}|_{2n}(S(k|_{2n})|_{2n}, q\|_{2n}) && (\text{Lem. 6.13.1}) \\
&= \textit{next-empty}|_{2n}(k|_{2n}, q\|_{2n}) && (test((k|_{2n})|_{2n}, q\|_{2n}),\ \text{Lem. 6.16.1})
\end{aligned}$$

6. By induction on the structure of q.

- $q = []$.
  Trivial.
- $q = inb(d, \ell, q')$.
  - $k = \ell$.

$$
\begin{array}{lll}
 & remove(k, inb(d,\ell,q'))\|_{2n} & \\
= & remove(k, q')\|_{2n} & (k = \ell) \\
= & remove(k|_{2n}, q'\|_{2n}) & \text{(i.h.)} \\
= & remove(k|_{2n}, add(d, \ell|_{2n}, q'\|_{2n})) & \text{(Lem. 6.18.5, } k = \ell) \\
= & remove(k|_{2n}, inb(d,\ell,q')\|_{2n}) &
\end{array}
$$

  - $k \neq \ell$.

    $test(\ell, q)$, hence $i \leq \ell < i + n$ by the assumption. Then $k|_{2n} \neq \ell|_{2n}$, using Lemma 6.13.4 and the fact that $i \leq k \leq i + n$.

$$
\begin{array}{lll}
 & remove(k, inb(d,\ell,q'))\|_{2n} & \\
= & inb(d, \ell, remove(k, q'))\|_{2n} & (k \neq \ell) \\
= & add(d, \ell|_{2n}, remove(k, q')\|_{2n}) & \\
= & add(d, \ell|_{2n}, remove(k|_{2n}, q'\|_{2n})) & \text{(i.h.)} \\
= & remove(k|_{2n}, add(d, \ell|_{2n}, q'\|_{2n})) & \text{(Lem. 6.20.4, } \; k|_{2n} \neq \ell|_{2n}) \\
= & remove(k|_{2n}, inb(d,\ell,q')\|_{2n}) &
\end{array}
$$

7. By induction on $k \dot{-} i$.

- $k \dot{-} i = 0$. By assumption $k \leq i$, so $i = k$. Then the lemma is trivial.
- $k \dot{-} i > 0$. Then $i < k$. Using Lemma 6.13.4 we get $i|_{2n} \neq k|_{2n}$. Also

$$
\begin{array}{lll}
 & test(j, remove(i, q)) & \\
\rightarrow & j \neq i & \text{(Lem. 6.14.3)} \\
\rightarrow & S(i) \leq j & (i \leq j)
\end{array}
$$

  Hence

$$\begin{aligned}
& release(i,k,q)\|_{2n} \\
&= release(S(i),k,remove(i,q))\|_{2n} && (i<k)\\
&= release|_{2n}(S(i)|_{2n},k|_{2n},remove(i,q)\|_{2n}) && \text{(above, i.h.)}\\
&= release|_{2n}(S(i|_{2n}),k|_{2n},remove(i,q)\|_{2n}) && \text{(Lem. 6.13.1)}\\
&= release|_{2n}(S(i|_{2n}),k|_{2n},remove(i|_{2n},q\|_{2n})) && \text{(Lem. 6.20.6)}\\
&= release|_{2n}(i|_{2n},k|_{2n},q\|_{2n}) && (i|_{2n}\neq k|_{2n})
\end{aligned}$$

8. By induction on the structure of q.
   - $q = []$.
     Trivial.
   - $q = inb(d', j, q')$.
     - $j < k$.
       Clearly $test(j, q)$. So $i \leq j < k$, by the assumption.
       Since $j < k$, $i \leq j < k \leq i + n$ and
       $i \leq k \leq i + n$, by Lemma 6.13.4,
       $j < k$ implies $j|_{2n} \neq k|_{2n}$.

$$\begin{aligned}
& add(d,k,inb(d',j,q'))\|_{2n} \\
&= inb(d',j,add(d,k,q'))\|_{2n} && (j<k)\\
&= add(d',j|_{2n},add(d,k,q')\|_{2n}) \\
&= add(d',j|_{2n},add(d,k|_{2n},q'\|_{2n})) && \text{(i.h.)}\\
&= add(d,k|_{2n},add(d',j|_{2n},q'\|_{2n})) && (j|_{2n}\neq k|_{2n},\ \text{Lem. 6.18.9})\\
&= add(d,k|_{2n},inb(d',j,q')\|_{2n})
\end{aligned}$$

     - $j \geq k$.

$$\begin{aligned}
& add(d,k,q)\|_{2n} \\
&= inb(d,k,remove(k,q))\|_{2n} && (j\geq k)\\
&= add(d,k|_{2n},remove(k,q)\|_{2n}) \\
&= add(d,k|_{2n},remove(k|_{2n},q\|_{2n})) && \text{(Lem. 6.20.6)}\\
&= add(d,k|_{2n},q\|_{2n}) && \text{(Lem. 6.20.1, Lem. 6.19.8)}
\end{aligned}$$

A.0.3 Invariants

In this section we prove the invariants in Lemma 6.21.

Proof.
For each case we only prove the first one, the second one is always the mirror, and is derived with a similar technique.

1. $h' \leq \mathit{next\text{-}empty}(\ell', q')$.

   h', ℓ', q' change only in summands F, H, I, K, M, O, P and Q. So we only need to check these summands.
   Among these, only F and H are non-trivial, because in other cases $h' := \mathit{next\text{-}empty}(\ell', q')$ or $h' := 0$.
   F: $q' := \mathit{add}(e, h, q')$;

   $h' \leq \mathit{next\text{-}empty}(\ell', q') \leq \mathit{next\text{-}empty}(\ell', \mathit{add}(e, h, q'))$
   (Lem. 6.18.2).
   H: $\ell' := S(\ell')$, $q' := \mathit{remove}(\ell', q')$; under condition $\mathit{test}(\ell', q')$;

   $h' \leq \mathit{next\text{-}empty}(\ell', q') = \mathit{next\text{-}empty}(S(\ell'), q') = \mathit{next\text{-}empty}(S(\ell'), \mathit{remove}(\ell', q'))$
   (Lem. 6.15.3).
2. $\ell \leq \mathit{next\text{-}empty}(\ell', q')$.

   Summands F, H and M need to be checked.
   F and H are provable with a similar strategy as the proof of Invariant 6.21.1.
   M: $\ell := h'$;

   $h' \leq \mathit{next\text{-}empty}(\ell', q')$ by Invariant 6.21.1.
3. $g' \neq 5 \to \ell \leq h'$.

   Summands I, J, K, L, M, O, P and Q need to be checked.

Summands K, M, P and Q are trivial, because in these cases $g' := 5$.

Summands J and L are also trivial since ℓ and h' do not alter.

I: $g' := 2$, $h' := \mathit{next\text{-}empty}(\ell', q')$;

By Invariant 6.21.2,
$\ell \leq \mathit{next\text{-}empty}(\ell', q')$.
O: $g' := 4$, $h' := \mathit{next\text{-}empty}(\ell', q')$;

Similar.

4. $\mathit{test}(i, q) \to i < m$.

   Summands A, M, P and Q need to be checked.
   A: $m := S(m)$, $q := \mathit{add}(d, m, q)$;

   $\mathit{test}(i, \mathit{add}(d, m, q)) \leftrightarrow i = m \vee \mathit{test}(i, q)$ using Lemma 6.18.3. Hence
   $i = m \vee i < m$ and therefore $i < S(m)$.
   M, P and Q: $q := \mathit{release}(\ell, h', q)$;

   $\mathit{test}(i, \mathit{release}(\ell, h', q)) \to \mathit{test}(i, q)$ (Lem. 6.14.5) $\to i < m$.
5. $(g = 3 \vee g = 4) \;\to\; h < m$.

   Summands A-G, S and T need to be checked.
   Among these only summands A, B and E are non-trivial, because in other cases $g \neq 3,\ 4$.
   A: $m := S(m)$;

   If $g \neq 0$, then $h < m < S(m)$.
   B: $g := 4$, $h := k$; under condition $\mathit{test}(k, q)$;

By Invariant 6.21.4, $test(k,q)$ implies $k < m$.
E: $g := 3$; under condition $g = 4$;

$g = 4$ implies $h < m$.

6. $test(i,q') \rightarrow i < m$.

Summands A, F and H need to be checked.
A: $m := S(m)$;

$test(i,q')$ implies $i < m < S(m)$.
F: $q' := add(e,h,q')$; under condition $g = 3$;

$g = 3$, so by Invariant 6.21.5, $h < m$. Hence,

$$\begin{array}{lcll} test(i, add(e,h,q')) & \leftrightarrow & (i = h \vee test(i,q')) & \text{(Lem. 6.18.3)} \\ & \rightarrow & (i = h \vee i < m) & \\ & \leftrightarrow & i < m & \end{array}$$

H: $q' := remove(\ell', q')$;

$test(i, remove(\ell', q')) \rightarrow test(i,q')$(Lem. 6.14.3) $\rightarrow i < m$.

7. $test(i,q') \rightarrow \ell' \leq i < \ell' + n$.

Summands F and H need to be checked.
F: $q' := add(e,h,q')$; under condition $\ell' \leq h < \ell' + n$;

$$\begin{array}{lcll} test(i, add(e,h,q')) & \leftrightarrow & i = h \vee test(i,q') & \text{(Lem. 6.18.3)} \\ & \rightarrow & i = h \vee \ell' \leq i < \ell' + n & \\ & \leftrightarrow & \ell' \leq i < \ell' + n & \end{array}$$

H: $\ell' := S(\ell')$, $q' := remove(\ell', q')$;

$$\begin{array}{lcll} test(i, remove(\ell', q')) & \leftrightarrow & test(i,q') \wedge i \neq \ell' & \text{(Lem. 6.14.3)} \\ & \rightarrow & \ell' \leq i < \ell' + n \wedge i \neq \ell' & \\ & \rightarrow & S(\ell') \leq i < S(\ell') + n & \end{array}$$

8. $\ell' \leq m$.

   Summands A and H need to be checked.

   A: $m := S(m)$;

   $\ell' \leq m < S(m)$.

   H: $\ell' := S(\ell')$; under condition $test(\ell', q')$;

   By Invariant 6.21.6, $test(\ell', q')$ implies $\ell' < m$.

   So $S(\ell') \leq m$.

9. $next\text{-}empty(\ell', q') \leq m$.

   By Invariant 6.21.8, $\ell' \leq m$.

   Furthermore, by Invariant 6.21.6, $\neg test(m, q')$.

   Hence, by Lemma 6.15.1,

   $next\text{-}empty(\ell', q') \leq m$.

10. $next\text{-}empty(\ell', q') \leq \ell' + n$.

    By Invariant 6.21.7, $\neg test(\ell' + n, q')$.

    Hence, by Lemma 6.15.1,

    $next\text{-}empty(\ell', q') \leq \ell' + n$.

11. $test(i, q) \rightarrow \ell \leq i$.

    Summands A, M, P and Q need to be checked.

    A: $q := add(d, m, q)$;

    By Invariant 6.21.2 and 6.21.9, $\ell \leq m$. So

$$
\begin{array}{rcll}
test(i, add(d, m, q)) & \leftrightarrow & i = m \vee test(i, q) & \text{(Lem. 6.18.3)} \\
& \rightarrow & i = m \vee \ell \leq i & \\
& \leftrightarrow & \ell \leq i &
\end{array}
$$

    M, P and Q: $\ell := h'$, $q := release(\ell, h', q)$;

$test(i, release(\ell, h', q)) \rightarrow test(i, q)$(Lem. 6.14.5) $\rightarrow \ell \leq i$.

12. $\ell \leq i < m \rightarrow test(i, q)$.

 Summands A, M, P and Q need to be checked.

 A: $m := S(m)$, $q := add(d, m, q)$;

 By Invariants 6.21.2 and 6.21.9, $\ell \leq m$. So

$$\begin{array}{lcll} \ell \leq i < S(m) & \rightarrow & \ell = m \vee \ell \leq i < m & \\ & \rightarrow & i = m \vee test(i, q) & \\ & \leftrightarrow & test(i, add(d, m, q)) & \text{(Lem. 6.18.3)} \end{array}$$

 M: $\ell := h'$, $q := release(\ell, h', q)$; under condition $g' = 1$;

 $g' = 1$, so by Invariant 6.21.3, $\ell \leq h'$. Hence,

$$\begin{array}{lcll} h' \leq i < m & \leftrightarrow & \ell \leq i < m \wedge \neg(\ell \leq i < h') & \\ & \rightarrow & test(i, q) \wedge \neg(\ell \leq i < h') & \\ & \leftrightarrow & test(i, release(\ell, h', q)) & \text{(Lem. 6.14.5)} \end{array}$$

 Summands P and Q hold similarly under condition $g' = 3$.

13. $m \leq \ell + n$.

 Summands A, M, P and Q need to be checked.

 A: $m := S(m)$; under condition $m < \ell + n$;

 Then $S(m) \leq \ell + n$.

 M: $\ell := h'$; under condition $g' = 1$;

 $g' = 1$, so by Invariant 6.21.3, $\ell \leq h'$.

 Hence, $m \leq \ell + n \leq h' + n$.

 Summands P and Q hold similarly, under condition $g' = 3$.

14. $(g = 3 \vee g = 4) \rightarrow next\text{-}empty(\ell', q') \leq h + n$.

Summands B-H, S and T need to be checked.

Among these only summands B, E and H are non-trivial, because in other cases $g \neq 3, 4$.

B: $g := 4$, $h := k$; under condition $test(k, q)$;

By Invariant 6.21.9, $next\text{-}empty(\ell', q') \leq m$.

By Invariant 6.21.13, $m \leq \ell + n$.

Since $test(k, q)$,

Invariant 6.21.11 yields $\ell \leq k$.

So $next\text{-}empty(\ell', q') \leq m \leq \ell + n \leq k + n$.

E: $g := 3$; under condition $g = 4$;

$g = 4$ implies $next\text{-}empty(\ell', q') \leq h + n$.

H: $\ell' := S(\ell')$, $q' := remove(\ell', q')$ under condition $test(\ell', q')$;

$$
\begin{array}{lll}
 & next\text{-}empty(S(\ell'), remove(\ell', q')) & \\
= & next\text{-}empty(S(\ell'), q') & \text{(Lem. 6.15.3)} \\
= & next\text{-}empty(\ell', q') & (test(\ell', q')) \\
\leq & h + n &
\end{array}
$$

15. $\ell' \leq i < h' \to test(i, q')$.

Summands F, H, I, K, M and O-Q need to be checked.

Among these only summands F, H, I and O are non-trivial, because in other cases $h' := 0$, and hence $\ell' \leq i < h'$ does not hold.

F: $q' := add(e, h, q')$;
$\ell' \leq i < h' \to test(i, q') \to test(i, add(e, h, q'))$ (Lem. 6.18.1). H: $\ell' := S(\ell')$, $q' := remove(\ell', q')$;
$S(\ell') \leq i < h' \leftrightarrow \ell' \leq i < h' \wedge i \neq \ell' \to test(i, q') \wedge i \neq \ell' \leftrightarrow test(i, remove(\ell', q'))$ (Lem. 6.14.3). I and O: $h' := next\text{-}empty(\ell', q')$;
By Lemma 6.15.1, $\ell' \leq i < next\text{-}empty(\ell', q') \to test(i, q')$.

16. $(g = 3 \vee g = 4) \wedge test(h, q) \ \to \ retrieve(h, q) = e$
Summands A-G, M, P, Q, S and T need to be checked. Among these only summands A,B,E and M are non-trivial, because in other cases $g \neq 3, 4$.

A: $q := add(d, m, q)$;

By Invariant 6.21.5, $g = 3 \vee g = 4$ implies $h < m$.
Hence, $retrieve(h, add(d, m, q)) = retrieve(h, q) = e$
(Lem. 6.18.4).
B: $g := 4$, $e := retrieve(k, q)$, $h := k$;

$retrieve(k, q) = retrieve(k, q)$ holds trivially.
E: $g := 3$; under condition $g = 4$;

If $test(h, q)$, then in view of $g = 4$, $retrieve(h, q) = e$.
M, P and Q: $q := release(\ell, h', q)$;

Let $(g = 3 \vee g = 4) \wedge test(h, release(\ell, h', q))$. By Lemma 6.14.5 $\neg(\ell \leq h < h')$.
Hence, by Lemma 6.14.6,
$retrieve(h, release(\ell, h', q)) = retrieve(h, q) = e$.

17. $test(i, q) \wedge test(i, q') \to retrieve(i, q) = retrieve(i, q')$.

Summands A, F, H, M, P and Q must be checked.
A: $q := add(d, m, q)$;

By Invariant 6.21.6, $test(i, q')$
implies $i \neq m$.
Hence

$$\begin{array}{lll} & test(i, add(d, m, q)) \wedge test(i, q') & \\ \leftrightarrow & test(i, q) \wedge test(i, q') & \text{(Lem. 6.18.3)} \\ \to & retrieve(i, q) = retrieve(i, q') & \\ \to & retrieve(i, add(d, m, q)) = retrieve(i, q') & \text{(Lem. 6.18.4)} \end{array}$$

F: $q' := add(e, h, q')$; under condition $g = 3$;

Let $test(i, q) \wedge test(i, add(e, h, q'))$.

CASE 1: $i \neq h$.

$$\begin{array}{lll} & test(i,q) \wedge test(i, add(e,h,q')) & \\ \rightarrow & test(i,q) \wedge test(i,q') & \text{(Lem. 6.18.3)} \\ \rightarrow & retrieve(i,q) = retrieve(i,q') & \\ \rightarrow & retrieve(i,q) = retrieve(i, add(e,h,q')) & \text{(Lem. 6.18.4)} \end{array}$$

CASE 2: $i = h$.

Then $retrieve(i, add(e,h,q')) = e$ using Lemma 6.18.4.

Suppose that $test(h,q)$.

Invariant 6.21.16 together with $g = 3$

yields $retrieve(h,q) = e$, which is $retrieve(i,q) = e$.

Therefore $retrieve(i, add(e,h,q')) = retrieve(i,q)$.

H: $q' := remove(\ell', q')$; [FGP+04]

By Lemma 6.14.3, $test(i, remove(\ell', q'))$

implies $i \neq \ell'$.

$$\begin{array}{lll} & test(i,q) \wedge test(i, remove(\ell', q')) & \\ \rightarrow & test(i,q) \wedge test(i,q') & \text{(Lem. 6.14.3)} \\ \rightarrow & retrieve(i,q) = retrieve(i,q') = retrieve(i, remove(\ell', q')) & \text{(Lem. 6.14.4)} \end{array}$$

M, P and Q: $q := release(\ell, h', q)$;

By Lemma 6.14.5,

$test(i, release(\ell, h', q))$ implies $\neg(\ell \leq i < h')$. Hence,

$$\begin{array}{lll} & test(i, release(\ell, h', q)) \wedge test(i,q') & \\ \rightarrow & test(i,q) \wedge test(i,q') & \text{(Lem. 6.14.5)} \\ \rightarrow & retrieve(i,q') = retrieve(i,q) & \\ \rightarrow & retrieve(i,q') = retrieve(i, release(\ell, h', q)) & \text{(Lem. 6.14.6)} \end{array}$$

18. $(g = 3 \vee g = 4) \wedge \mathit{test}(h, q') \rightarrow \mathit{retrieve}(h, q') = e$.

    Summands B-H need to be checked.
    Among these only summands
    B, E and H are non-trivial,
    because in other cases $g \neq 3, 4$.
    B: $g = 4$, $e := \mathit{retrieve}(k, q)$, $h := k$; under condition $\mathit{test}(k, q)$;

    If $\mathit{test}(k, q')$, then by Invariant 6.21.17,
    $\mathit{retrieve}(k, q') = \mathit{retrieve}(k, q)$.
    E: $g := 3$; under condition $g = 4$;

    If $\mathit{test}(h, q')$, then in view of $g = 4$, $\mathit{retrieve}(h, q') = e$.
    H: $q' := \mathit{remove}(\ell', q')$;

    Let $g = 3 \vee g = 4$ and $\mathit{test}(h, \mathit{remove}(\ell', q'))$. By Lemma 6.14.3,
    $h \neq \ell'$. Hence, by Lemma 6.14.4,
    $\mathit{retrieve}(h, \mathit{remove}(\ell', q')) = \mathit{retrieve}(h, q') = e$.

19. $\ell \leq i \wedge j \leq \mathit{next\text{-}empty}(i, q') \;\rightarrow\; q[i..j\rangle = q'[i..j\rangle$.
    We apply induction on $j \dot{-} i$.

    - If $i \geq j$, then $q[i..j\rangle = \langle\rangle = q'[i..j\rangle$.
    - If $i < j$, then
      $i < \mathit{next\text{-}empty}(i, q')$, therefore $\mathit{test}(i, q')$, and hence $i < m$
      by Invariant 6.21.6.
      Now $\ell \leq i < m$, so by Invariant 6.21.12
      $\mathit{test}(i, q)$. Hence,

$$
\begin{array}{rcll}
q[i..j\rangle l & = & \mathit{inb}(\mathit{retrieve}(i, q), q[S(i)..j\rangle) & \\
 & = & \mathit{inb}(\mathit{retrieve}(i, q), q'[S(i)..j\rangle) & \text{(i.h.)} \\
 & = & \mathit{inb}(\mathit{retrieve}(i, q'), q'[S(i)..j\rangle) & \text{(Inv. 6.21.17)} \\
 & = & q'[i..j\rangle. &
\end{array}
$$

■

Bibliography

[Ack54] W. Ackermann. *Solvable Cases of the Decision Problem.* Studies in Logic and the Foundations of Mathematics. North-Holland, 1954.

[Bau00] P. Baumgartner. FDPLL – A First-Order Davis-Putnam-Logeman-Loveland Procedure. In D.A. McAllester, editor, *Proc. 17th Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence 1831, pages 200–219. Springer-Verlag, 2000.

[BB04] C. Barrett and S. Berezin. CVC lite: A new implementation of the cooperating validity checker. In R. Alur and D. Peled, editors, *Proc. 16th Conference on Computer Aided Verification*, Lecture Notes in Computer Science 3114, pages 515–518, 2004.

[BDL96] C.W. Barrett, D.L. Dill, and J.R. Levitt. Validity checking for combinations of theories with equality. In M.K. Srivas and A. Camilleri, editors, *Proc. 1st conference on Formal Methods in Computer Aided Design*, Lecture Notes in Computer Science 1166, pages 187–201. Springer-Verlag, 1996.

[BDS00] C.W. Barrett, D.L. Dill, and A. Stump. A framework for cooperating decision procedures. In D.A. McAllester, editor, *Proc. 17th Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence 1831, pages 79–98. Springer-Verlag, 2000.

[BFG$^+$01] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I.A. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. 13th Conference on Computer Aided Verification*, Lecture Notes in Computer Science 2102, pages 250–254. Springer-Verlag, 2001.

[BFG$^+$05] B. Badban, W.J. Fokkink, J.F. Groote, J. Pang, and J.C. van de Pol. Verifying a sliding window protocol in μCRL and PVS. *Formal Aspects of Computing*, 17(3):342–388, 2005.

[BG94a] M.A. Bezem and J.F. Groote. A correctness proof of a one bit sliding window protocol in μCRL. *The Computer Journal*, 37(4):289–307, 1994.

[BG94b] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. In B. Jonsson and J. Parrow, editors, *Proc. 5th Conference on Concurrency Theory*, Lecture Notes in Computer Science 836, pages 401–416. Springer-Verlag, 1994.

[BGV99] R.E. Bryant, S. German, and M.N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In N. Halbwachs and D. Peled, editors, *Proc. 11th Conference on Computer Aided Verification*, Lecture Notes in Computer Science 1633, pages 470–482. Springer-Verlag, 1999.

[BGV01] R.E. Bryant, S. German, and M.N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(4):93–134, 2001.

[BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[Bry92] R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[BvdP04] B. Badban and J.C. van de Pol. An algorithm to verify formulas by means of $(0, S, =)$-BDDs. In H. Rabiee, editor, *Proc. 9th Annual Computer Society of Iran Computer Conference*, pages 54–63. Iranian Computer Society, 2004.

[BvdP05] B. Badban and J.C. van de Pol. Zero, successor and equality in binary decision diagrams. *Annals of Pure and Applied Logic*, 133(1-3):101–123, 2005.

[BvdPTZ04a] B. Badban, J.C. van de Pol, O. Tveretina, and H. Zantema. Generalizing DPLL and satisfiability for equalities. Technical Report SEN-R0407, CWI, 2004.

[BvdPTZ04b] B. Badban, J.C. van de Pol, O. Tveretina, and H. Zantema. Solving satisfiability of ground term algebras using DPLL and unification. In *Proc. 18th Workshop on Unification*, pages 21–36, Cork, Ireland, 2004. Stichting Mathematisch Centrum.

[BW94] A.J.C. Bik and H.A.G. Wijshoff. Implementation of Fourier-Motzkin elimination. Technical Report 94-42, Dept. of Computer Science, Leiden University, 1994.

[CHdV03] D. Chkliaev, J. Hooman, and E. de Vink. Verification and improvement of the sliding window protocol. In H. Garavel and J. Hatcliff, editors, *Proc. 9th Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 2619, pages 113–127. Springer-Verlag, 2003.

[CK74] V.G. Cerf and R.E. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, COM-22:637–648, 1974.

[CL89] H. Comon and P. Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7(3–4):371–425, 1989.

[CO91] R. Cardell-Oliver. Using higher order logic for modeling real-time protocols. In J. Diaz and F. Orejas, editors, *Proc. 4th Joint Conference on Theory and Practice of Software Development*, Lecture Notes in Computer Science 494, pages 259–282. Springer-Verlag, 1991.

[Col84] A. Colmerauer. Equations and inequations on finite and infinite trees. In ICOT staff, editor, *Proc. Conference on Fifth Generation Computer Systems*, pages 85–99. North-Holland, 1984.

[Der87] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1–2):69–115, 1987.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[FGP+04] W.J. Fokkink, J.F. Groote, J. Pang, B. Badban, and J.C. van de Pol. Verifying a sliding window protocol in μCRL. In S. Maharaj C. Rattray and C. Shankland, editors, *Proc. 10th Conference on Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science 3116, pages 148–163. Springer-Verlag, 2004.

[FORS01] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated canonizer and solver. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. 13th Conference on Computer Aided Verification*, Lecture Notes in Computer Science 2102, pages 246–249. Springer-Verlag, 2001.

[FP03] W.J. Fokkink and J. Pang. Cones and foci for protocol verification revisited. In A.D. Gordon, editor, *Proc. 6th Conference on Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science 2620, pages 267–281. Springer-Verlag, 2003.

[FPvdP05] W.J. Fokkink, J. Pang, and J.C. van de Pol. Cones and foci: A mechanical framework for protocol verification. Submitted to: *Formal Methods in System Design*, 2005.

[GHN+04] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In R. Alur and D. Peled, editors, *Proc. 16th Conference on Computer Aided Verification*, Lecture Notes in Computer Science 3114, pages 175–188, 2004.

[GK95] J.F. Groote and H.P. Korver. Correctness proof of the bakery protocol in μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Proc. 1st Workshop on Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 63–86. Springer-Verlag, 1995.

[GL99] P. Godefroid and D.E. Long. Symbolic protocol verification with Queue BDDs. *Formal Methods and System Design*, 14(3):257–271, 1999.

[GP94] J.F. Groote and A. Ponse. Proof theory for μCRL: A language for processes with data. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Proc. Workshop on Semantics of Specification Languages*, Workshops in Computing Series, pages 231–250. Springer-Verlag, 1994.

[GP95] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Proc. 1st Workshop on Algebra*

of Communicating Processes '94, Workshops in Computing Series, pages 26–62. Springer-Verlag, 1995.

[GPU01] J.F. Groote, A. Ponse, and Y.S. Usenko. Linearization in parallel pCRL. *Journal of Logic and Algebraic Programming*, 48(1-2):39–72, 2001.

[GR01] J.F. Groote and M.A. Reniers. Algebraic process verification. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 17, pages 1151–1208. Elsevier, 2001.

[Gro87] R.A. Groenveld. Verification of a sliding window protocol by means of process algebra. Technical Report P8701, University of Amsterdam, 1987.

[GS01] J.F. Groote and J. Springintveld. Focus points and convergent process operators. a proof strategy for protocol verification. *Journal of Logic and Algebraic Programming*, 49(1-2):31–60, 2001. Also appeared as Logical Preprint 142, Department of Philosophy, Utrecht University, 1993.

[GSZA98] A. Goel, K. Sajid, H. Zhou, and A. Aziz. BDD based procedures for a theory of equality with uninterpreted functions. In A. J. Hu and M.Y. Vardi, editors, *Proc. 10th Conference on Computer Aided Verification*, Lecture Notes in Computer Science 1427, pages 244–255. Springer-Verlag, 1998.

[GvdP00] J.F. Groote and J.C. van de Pol. Equational binary decision diagrams. In M. Parigot and A. Voronkov, editors, *Proc. 7th Conference on Logic for Programming and Automated Reasoning*, Lecture Notes in Artificial Intelligence 1955, pages 161–178. Springer-Verlag, 2000.

[GZ03] J.F. Groote and H. Zantema. Resolution and binary decision diagrams cannot simulate each other polynomially. *Discrete Applied Mathematics* 130(2):157-171, 2003.

[Hai82] B.T. Hailpern. *Verifying Concurrent Processes Using Temporal Logic.* Lecture Notes in Computer Science 129. Springer-Verlag, 1982.

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[Hol97] G.J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[JN00] B. Jonsson and M. Nilson. Transitive closures of regular relations for verifying infinite-state systems. In S. Graf and M. Schwartzbach, editors, *Proc. 6th Conference on Tools and Algorithms for Construction and Analysis of Systems*, Lecture Notes in Computer Science 1785, pages 220–234. Springer-Verlag, 2000.

[Jon87] B. Jonsson. *Compositional Verification of Distributed Systems*. PhD thesis, Department of Computer Science, Uppsala University, 1987.

[Kai97] R. Kaivola. Using compositional preorders in the verification of sliding window protocol. In O. Grumberg, editor, *Proc. 9th Conference on Computer Aided Verification*, Lecture Notes in Computer Science 1254, pages 48–59. Springer-Verlag, 1997.

[Knu81] D.E. Knuth. Verification of link-level protocols. *BIT*, 21:21–36, 1981.

[Lat01] T. Latvala. Model checking LTL properties of high-level Petri nets with fairness constraints. In J.M. Colom and M. Koutny, editors, *Proc. 21st Conference on Application and Theory of Petri Nets*, Lecture Notes in Computer Science 2075, pages 242–262. Springer-Verlag, 2001.

[LEW96] J. Loeckx, H.D. Ehrich, and M. Wolf. *Specification of Abstract Data Types.* Wiley/Teubner, 1996.

[LMM87] J-L. Lassez, M. J. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann Publishers, 1987.

[Mah88] M.J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proc. 3rd Annual Symposium on Logic in Computer Science*, pages 348–357. IEEE Computer Society Press, 1988.

[McC62] J. McCarthy. Checking mathematical proofs by computer. In *Proc. Symposium on Recursive Function Theory*, pages 219–227. American Mathematical Society, 1962.

[MLAH99] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. In J. Flum and M. Rodríguez-Artalejo, editors, *Proc. 13th Workshop on Computer Science Logic*, Lecture Notes in Computer Science 1683, pages 111–125. Springer-Verlag, 1999.

[MMZ+01] M. Moskewicz, C. Madigan, Y. Zhaod, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proc. 39th Design Automation Conference*, pages 530–535. ACM, 2001.

[MV91] E. Madelaine and D. Vergamini. Specification and verification of a sliding window protocol in LOTOS. In E. Knuth and L.M. Wegner, editors, *Proc. 4th Conference on Formal Description Techniques for Distributed Systems and Communication Protocols,* IFIP Transactions (C-2), pages 495–510. North-Holland, 1991.

[NO80] G. Nelson and D.C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.

[NO05] R. Nieuwenhuis and A. Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In K.Etessami and S.K. Rajamani, editors, *Proc. 17th Conference on Computer Aided Verification*, Lecture Notes in Computer Science 3576, pages 321–334. Springer-Verlag, 2005.

[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Proc. 8th Conference on Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 411–414. Springer-Verlag, 1996.

[Par81] D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proc. 5th GI-Conference on Theoretical Computer Science*, Lecture Notes in Computer Science 104, pages 167–183. Springer-Verlag, 1981.

[Pic03] R. Pichler. On the complexity of equational problems in CNF. *Journal of Symbolic Computation*, 36(1-2):235–269, 2003.

[Pra70] V. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, 1970.

[PRSS99] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small domains instantiations. In N. Halbwachs and D. Peled, editors, *Proc. 11th Conference on Computer Aided Verification*, Lecture Notes in Computer Science 1633, pages 455–469. Springer-Verlag, 1999.

[PS91] K. Paliwoda and J.M. Sanders. An incremental specification of the sliding-window protocol. *Distributed Computing*, 5(2):83–94, 1991.

[RE99] C. Röckl and J. Esparza. Proof-checking protocols using bisimulations. In J.C.M. Baeten and S. Mauw, editors, *Proc. 10th Conference on Concurrency Theory*, Lecture Notes in Computer Science 1664, pages 525–540. Springer-Verlag, 1999.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–49, 1965.

[RRSV87] J.L. Richier, C. Rodriguez, J. Sifakis, and J. Voiron. Verification in Xesar of the sliding window protocol. In H. Rudin and C.H. West, editors, *Proc. 7th Conference on Protocol Specification, Testing and Verification*, pages 235–248. North-Holland, 1987.

[Rus01] V. Rusu. Verifying a sliding-window protocol using PVS. In M. Kim, B. Chin, S. Kang, and D. Lee, editors, *Proc. 21st Conference on Formal Techniques for Networked and Distributed Systems,* IFIP Conference Proceedings 197, pages 251–268. Kluwer Academic, 2001.

[SBD02] A. Stump, C.W. Barrett, and D.L. Dill. CVC: A cooperating validity checker. In J.C. Godskesen, editor, *Proc. 14th Conference on Computer Aided Verification*, Lecture Notes in Computer Science 2404, pages 500–505. Springer-Verlag, 2002.

[SBLS99] K. Stahl, K. Baukus, Y. Lakhnech, and M. Steffen. Divide, abstract, and model-check. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Proc. 6th SPIN Workshop on Practical Aspects of Model Checking*, Lecture Notes in Computer Science 1680, pages 57–76. Springer-Verlag, 1999.

[Sch91] A.A. Schoone. *Assertional Verification in Distributed Computing.* PhD thesis, Utrecht University, 1991.

[Sho78] R.E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583–585, 1978.

[SK00] M.A. Smith and N. Klarlund. Verification of a sliding window protocol using IOA and MONA. In T. Bolognesi and D. Latella, editors, *Proc. 20th Joint Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*, pages 19–34. Kluwer Academic Publishers, 2000.

[SLB02] S.A. Seshia, S. Lahiri, and R.E. Bryant. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In E. Brinksma and K.G. Larsen, editors, *Proc. 14th Conference on Computer Aided Verification*, Lecture Notes in Computer Science 2404, pages 78–92. Springer-Verlag, 2002.

[SR02] N. Shankar and H. Rueß. Combining Shostak theories. In S. Tison, editor, *Proc. 13th Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science 2378, pages 1–18. Springer-Verlag, 2002.

[SSB02] O. Strichman, S.A. Seshia, and R.E. Bryant. Deciding separation formulas with SAT. In E. Brinksma and K.G. Larsen, editors, *Proc. 14th Conference on Computer Aided Verification*, Lecture Notes in Computer Science 2404, pages 209–222. Springer-Verlag, 2002.

[Ste76] N.V. Stenning. A data transfer protocol. *Computer Networks*, 1:99–110, 1976.

[Str02] O. Strichman. On solving Presburger and linear arithmetic with SAT. In M.D. Aagaard and J.W. O'Leary, editors, *Proc. 4th Conference on Formal Methods in Computer Aided Design*, Lecture Notes in Computer Science 2517, pages 160–170. Springer-Verlag, 2002.

[Tan81] A.S. Tanenbaum. *Computer Networks*. Prentice Hall, 1981.

[Ter03] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[Tin02] C. Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *Proc. Conference on Logics in Artificial Intelligence*, Lecture Notes in Computer Science 2424, pages 308–319. Springer-Verlag, 2002.

[Tve05] O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. PhD thesis, Eindhoven University of Technology, 2005.

[Vaa86] F.W. Vaandrager. Verification of two communication protocols by means of process algebra. Technical Report Report CS-R8608, CWI, 1986.

[vdPT05] J.C. van de Pol and O. Tveretina. A bdd-representation for the logic of equality and uninterpreted functions. In J. Jedrzejowicz and A. Szepietowski, editors, *Mathematical Foundations of Computer Science*, volume 3618 of *Lecture Notes in Computer Science*, pages 769–780. Springer-Verlag, 2005.

[vdS95] J.L.A. van de Snepscheut. The sliding window protocol revisted. *Formal Aspects of Computing*, 7(1):3–170, 1995.

[vG94] R.J. van Glabbeek. What is branching time and why to use it? *The Concurrency Column, Bulletin of the EATCS*, 53:190–198, 1994.

[vGW96] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.

[vW92] J.J. van Wamel. A study of a one bit sliding window protocol in ACP. Technical Report P9212, University of Amsterdam, 1992.

[Zan03] H. Zantema. Termination. In M.A. Bezem, J.W. Klop, and R.C. de Vrijer, editors, *Term Rewriting Systems*, chapter 6, pages 181–259. Cambridge University Press, 2003.

[ZvdP01] H. Zantema and J.C. van de Pol. A rewriting approach to binary decision diagrams. *Journal of Logic and Algebraic Programming*, 49(1-2):61–86, 2001.

Summary

In the first part of this thesis we focus on extensions of existing theorem provers, to larger theories. We consider two best known techniques BDD and DPLL methods. Our target is to have some incremental techniques for deciding theories of quantifier-free first-order logic.

The first approach is based on the BDD method. We introduce a theorem prover to decide satisfiability of equality logic with zero and successor. The idea is to present the input formula as a BDD, then transform the BDD formula to an equivalent Ordered BDD. This is done in such a way that on the resulting Ordered BDD, tautology and satisfiability can be checked in constant time.

Our second approach is based on the DPLL procedure. We generalize the procedure to a theorem prover, called GDPLL, which decides the theory of ground term algebras. In order to illustrate how (dissimilar) these two methods, Ordered BDD and GDPLL, behave, we bring an example of their common theory. Further, we extend the underlying logic of GDPLL to the theory of ground term algebras with recognizers. LISP list structure and PVS datatype declarations are two examples which use recognizers. We demonstrate the technique on some formulas in the LISP language.

The second part of the thesis is on application and utilizing verification techniques. Our interest is to depict reliability of existing protocols. The sliding window protocol is a core protocol in Transmission Control Protocols. TCPs are extensively used in the Internet. We verify a two-sided sliding window protocol. This protocol has the acknowledgements piggybacked on data. This way acknowledgements take a free ride in the channel. As a result the available bandwidth is used better. We present a specification of sliding window protocol with piggy backing in μCRL, and then verify the specification with the PVS theorem prover.

Samenvatting

In het eerste gedeelte van dit proefschrift richten we ons op extensies van bestaande theorem provers voor uitbreidingen van propositielogica. We beschouwen twee van de bekendste technieken, BDD en DPLL methoden. Ons doel is om enkele incrementele technieken te hebben voor beslisbare theorieën van kwantor-vrije eerste orde logica.

De eerste aanpak is gebaseerd op de BDD methode. We introduceren een theorem prover om de vervulbaarheid van gelijkheidslogica met nul en opvolger te bepalen. Het idee is om de invoerformule als een BDD te presenteren en vervolgens de BDD formule te transformeren naar een equivalente geordende BDD. Dit wordt dusdanig gedaan dat op de resulterende geordende BDD tautologie en satisfiability kan worden gecontroleerd in constante tijd.

Onze tweede aanpak is gebaseerd op de DPLL procedure. We generaliseren de procedure naar een theorem prover, genaamd GDPLL, die de theorie van gesloten term-algebra's beslist. Om te illustreren hoe (verschillend) deze twee methoden, geordende BDD en GDPLL, zich gedragen, tonen we een voorbeeld van hun gemeenschappelijke theorie. Vervolgens breiden we de onderliggende logica van GDPLL uit naar de theorie van gesloten term-algebra's met herkenners. LISP lijststructuren en PVS datatype declaraties zijn twee voorbeelden die herkenners gebruiken. We demonstreren de techniek op enkele formules in de LISP taal.

Het tweede gedeelte van het proefschrift gaat over toepassing en gebruik van verificatietechnieken. Ons doel is om betrouwbaarheid van bestaande protocollen aan te tonen. Het sliding window protocol is een kernprotocol in Transmission Control Protocollen. TCPs worden breedschalig gebruikt op het internet. We verifiëren een tweezijdig sliding window protocol. Dit protocol heeft de bevestigingen gepiggybackt op data. Op deze wijze gaan bevestigingen gratis mee over het kanaal. Een resultaat hiervan is dat de beschikbare bandbreedte beter gebruikt wordt. We presenteren een specificatie van het sliding window protocol met piggy backing in μCRL, en verifiëren dan de specificatie met de PVS theorem prover.

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-01

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-02

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-03

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-04

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-05

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-06

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-07

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-08

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-09

A.M.G. Peeters. *Single-Rail Handshake Circuits*. Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics*. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context*. Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types*. Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics*. Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution*. Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, UL. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection*. Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Scheduler Optimization in Real-Time Distributed Databases*. Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection*. Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences*. Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking*. Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols*. Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents*. Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ*. Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing state space reduction techniques for model checking*. Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*.

Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13