UNIVERSITÀ DEGLI STUDI DI ROMA
"TOR VERGATA"

FACOLTÀ DI INGEGNERIA

DOTTORATO DI RICERCA IN
INFORMATICA ED INGEGNERIA DELL'AUTOMAZIONE

XX CICLO

Transformation Techniques for Constraint Logic Programs with Applications to Protocol Verification

Valerio Senni

Tutor:	prof. Alberto Pettorossi
	dott. Maurizio Proietti
Coordinatore:	prof. Daniel Pierre Bovet

Acknowledgements

I am very grateful to my advisor Alberto Pettorossi who first sparked my interest for research in the field of formal methods and theoretical computer science. I have learned a lot from him: the joy of studying, the search for the 'entire picture' in my scientific work, the trust in the reality and in our investigations. Our occasional trips to the mountains (the Sibillini), trying to get to the top, leave a sign in my memories as a metaphor of all this.

I am deeply indebted to my co-advisor Maurizio Proietti. His patience in sharing with me his scientific experience is invaluable. He spent a lot of effort in teaching me the methods of scientific research. I have learned that nothing is most important for research and for life than avoiding to be schematic, looking for new ways, being curious, and keep asking for reasons.

Collaborating with Alberto and Maurizio in these years have been really precious to me. I have also grown as a member of a research group: this is as important as breathing for a young researcher and it is a privilege that sadly not so many students have.

I acknowledge the University of Rome "Tor Vergata" and, in particular, the Department of Informatics, Systems and Production and professor Daniel Pierre Bovet for their support during my PhD studies. I also thank the IASI institute of the italian National Research Council, for offering me a desk space during my frequent visits to Maurizio Proietti.

On a personal level, I want to thank Santina for her support in these years. When my PhD course began we were still engaged and now she has become my wife. This is a simple and deep sign that we are growing and building together. I also want to thank my parents. They always helped me and supported me in my choices with respect and trust. This work has been possible also for their continuous and reliable presence.

Abstract

The contribution of this thesis consists in the extension of the techniques for the transformation of constraint logic programs and the development of methods for the application of these techniques to the proof of temporal properties of parameterized protocols.

In particular, we first introduce a method for proving automatically the total correctness of an unfold/fold transformation by solving linear equations and inequations over the natural numbers.

We also propose a transformation-based method for proving first order properties of constraint logic programs which manipulate finite lists of real or rational numbers.

Then, we extend the standard folding transformation rule by introducing two variants of this rule. The first variant combines the folding rule with the clause splitting rule for obtaining a more powerful folding rule. The second variant is tailored to the elimination of the existential variables occurring in a clause. For the standard folding rule and its two variants we develop the corresponding algorithms for automating their application.

Finally, we propose a program transformation framework for proving temporal properties of parameterized protocols. Using this framework we encode the protocols and the temporal properties we want to prove as logic programs, and then we use the unfold/fold transformation technique for proving whether or not the properties holds.

Contents

Chapter 1

Introduction

In this thesis we study various techniques for the transformation of logic programs and constraint logic programs to the aim of proving properties of software systems and, in particular, properties of parameterized protocols.

Program Transformation, introduced in [17, 79], has been long recognized as a powerful technique for program development. It has also been shown that unfold/fold transformations can be used to prove program properties such as equivalences of functions defined by recursive equation programs [20, 36], properties of relations defined by logic programs [54, 55], and temporal properties of protocols [26, 67].

The *Constraint Logic Programming* [32] paradigm extends the usual logic programming framework [43] by: (i) allowing constraints over a generic domain $\mathcal{D}$ and (ii) integrating the resolution-based operational semantics with efficient algorithms specifically developed for the constraint domain under consideration.

The integration of the research fields on program transformation and constraint logic programming has been the topic of many investigations and, in particular, reveals to be very prolific in the area of parameterized protocols verification because, in that field, one can take advantage of both the deduction capabilities of program transformation and the expressiveness and efficiency of constraint logic programming.

The contributions of this thesis consist in the extension of the transformation techniques available for constraint logic programs and in the development of methods for the application of these techniques to the proof of temporal properties of parameterized protocols.

1.1 Correctness of Software Systems

In order to deal with complexity software systems, several research efforts have been devoted to the development of methods for proving correctness. The general problem of the *correctness* of a software system is one of the earliest and fundamental problems in Computer Science, and consists in establishing whether or not a software system exhibits an 'expected' *behaviour*. We can rephrase the problem as follows. We identify a software system with its *specifications* and we want to *verify* whether or not a desired *property* holds for these specifications. Among the various approaches, a large research field is the one which uses formal methods to give a precise specification of a system and of the properties of interest.

The goal of the research based on formal methods is: (i) to give a mathematical representation of a system specifications, that is, a *mathematical theory*, (ii) to provide a *formal language* to express properties, and (iii) to develop *theorem proving* techniques to establish whether a formula holds or not in the given theory. Since the problem of theorem proving is in general undecidable [49], a fully mechanical, complete method cannot be provided.

In particular, our method adopts the logic programming paradigm to represent the system specifications and a transformational approach to prove first order properties of logic programs. As a consequence, in our view, the problem of proving the correctness of software systems reduces to the problem of proving first order properties of logic programs.

The choice of logic programming is motivated by several useful features of this programming paradigm. First, logic programming has both a clear semantics and a practical notion of computation [43]. This entails that specifications written as a logic program are executable. That is, we may have an executable model of the system under investigation. Second, logic programming is very expressive and it allows a declarative programming

style. These features have been enhanced since logic programming have been extended to constraint logic programming and several efficient constraint resolution algorithms have been integrated with the resolution-based operational semantics. Finally, both the program representing the system under investigation and the first order property to be proved are expressed within the same logical language and this allows us to manipulate both the property and the specifications by using the same tools.

1.2 Proving Properties of Logic Programs

Our approach to the problem of proving properties of logic programs uses the program transformation methodology, based on rules and strategies [17, 52, 79], as a deduction method. Program transformation consists in the process of deriving new programs from old ones by applying semantics preserving transformation rules. Therefore, if the initial program P_0 has a model $M(P_0)$ and, by transformation, we construct the sequence $P_0, \ldots, P_n$ of programs by applying semantics preserving transformation rules, then the model $M(P_n)$ of the final program P_n is equal to $M(P_0)$. For example, let a property φ be denoted by a predicate *prop* defined in the program P_0. If the clause $prop \leftarrow$ belongs to the final program P_n, then the property denoted by the predicate *prop* holds in $M(P_n)$. Since the transformation rules are semantics preserving, we have that *prop* holds also in $M(P_0)$. In order to obtain an effective deduction method based on this transformation technique, we need to develop strategies that guide the application of the transformation rules in a suitable way.

In this thesis we will illustrate, in particular, how the general setting of program transformation can be applied to the problem of protocol verification.

1.3 Verification of Parameterized Protocols

Protocols are sets of rules which should be obeyed by processes that interact in a concurrent environment for guaranteeing that their interactions enjoy some desirable properties such as the absence of deadlocks and the

mutually exclusive access to shared resources. To this purpose, many sophisticated protocols have been designed and proposed in the literature. Usually the number of interacting processes is not known in advance and protocol specifications are parameterized with respect to the number of processes. The problem of verifying properties of parameterized protocols is in general undecidable [4] and thus we cannot expect to develop a fully mechanical, complete method.

In order to develop a transformational technique for verifying properties of protocols, we first need to provide a formalization of the notion of protocol and to define a method for encoding a protocol into a logic program. Furthermore, we need to adopt a specific logic in order to express the properties of interest.

To give a formalization of the notion of protocol, let us first observe that we are not interested in the 'local computations' performed by every single process running in our concurrent environment. On the contrary, we are interested in describing the states of the processes and the communications among these processes. These aspects, which may be called the *observables* of the considered systems, are captured by the notion of *state transition system*. A state transition system is a pair $\langle S, T \rangle$ where S is the set of *states* of the system and T is the set of possible *transitions* among these states.

The notion of state transition system has been first developed in the field of Model Checking [18], which is a very powerful technique for proving properties of *finite* state protocols. In particular, for the class of finite state protocols a number of temporal logics has been defined for which the problem of verification is decidable. From the field of model checking we borrow both the notion of state transition system and the logic used for expressing the properties of interest, but the general problem of verifying properties of parameterized protocols remains undecidable even for the chosen logic.

In particular, we are interested in verifying properties of the evolution in time of protocols. In order to express these properties, we will adopt the *Computational Tree Logic* [18] (CTL, for short), which is a branching time temporal logic. CTL does not consider a time variable explicitly, but represents time via sequences of states and branching of these sequences.

Our method does not necessarily require the use of the CTL logic, but it can be adapted to the use of different logics, that can be more expressive or tailored to a particular class of properties.

Now we briefly outline our verification technique. A more in depth discussion of the technical aspects of this technique will be given in the following chapters.

Our verification technique provides: (i) a method for encoding the state transition system specification of a protocol as a logic program P, (ii) a method for encoding a CTL formula φ as a set F of clauses, and (iii) a transformation strategy which transforms the program $P \cup F$ in order to prove whether or not the formula φ holds in the perfect model of the program P.

A straightforward advantage of this approach is that several techniques developed in the field of program transformation can be used or adapted to the verification of properties of protocols.

In the following section we briefly overview the content of each chapter of the thesis.

1.4 Overview of the Thesis

In Chapter 2 we address the problem of proving that a program transformation is totally correct, that is, in the case of definite logic programs, it preserves the least Herbrand model semantics. We propose a method for proving in an automatic way that the program transformations obtained by applying the definition introduction, folding, unfolding, and goal replacement rules are totally correct. Our method associates with each clause an annotation, consisting in an unknown ranging over the natural numbers and, at every application of the transformation rules, modifies both the program and the clauses annotations. In particular, during the construction of a transformation sequence, every rule application generates some constraints, which are linear equations and inequations, that relate the unknowns of the clauses involved in the transformation. The satisfiability of this set of constraints ensures the correctness of the transformation. Fi-

nally, we consider transformation sequences constructed by using also the goal replacement rule and we present a proof method which constructs the constraints needed for a correct application of this rule.

Our technique for proving the correctness of program transformations is related to several approaches proposed in the literature, which associate a well-founded ordering to the proof trees of the atoms that belong to the least Herbrand model of the program to be transformed. This ordering ensures that, when transforming a program P into a program T, we preserve the program semantics, that is, the least Herbrand models of P and T are equal. This ordering, however, is obtained by fixing some clause measures in advance and independently of the transformation that is actually performed. Our method, instead, computes these measures automatically during the transformation. We also propose some experimental results obtained by using an implementation of this technique in the MAP transformation system [46]. The results illustrated in this chapter have been published in [59].

In Chapter 3 we propose a method for proving first order properties of constraint logic programs whose constraints are linear equations and inequations over the rational or real numbers. This method is based on the encoding of the given first order formula into a stratified constraint logic program and on the application of a suitable transformation strategy that preserves the perfect model of the program. The goal of our strategy is the elimination of the existential variables occurring in the given program in order to transform it into an equivalent program such that the truth or falsity of the formula can be checked by looking at the syntactical properties of the program clauses. Since, in general, the first order properties of the class of programs we consider are undecidable, our strategy is necessarily incomplete. At the end of this Chapter 3, we present some experimental results obtained by using a prototype implementation of this method in the MAP transformation system. This implementation allowed us to prove some non-trivial properties of constraint logic programs that manipulate finite lists of rational or real numbers. The results illustrated in this chapter have been published in [57].

In Chapter 4 we consider the standard folding transformation rule for

constraint logic programs and propose an algorithm based on linear algebra and term rewriting techniques to automate its application. Moreover, we extend the definition of the folding rule by proposing two variants of this rule. In the first variant we integrate the clause splitting rule with the folding rule in order to obtain a more powerful rule. The second variant of the folding rule is tailored to the elimination of existential variables. For both variants we propose sound and complete algorithms for their application, based on techniques similar to those used to develop the algorithm for the standard folding rule. Some examples of application of the folding rule and his variants illustrate the algorithms proposed. The results illustrated in this chapter have been submitted for publication [75].

Finally, in Chapter 5 we define a framework based on a first order theory of arrays and linear constraints over the rational or real numbers which allows us to formally specify parameterized protocols, and to prove their properties. We start by recalling the parameterized Peterson's protocol [51] for mutual exclusion, which is used throughout the chapter as a working example. Then, we present our specification method for protocols which makes use of an extension of stratified logic programs where bodies of clauses may contain first order formulas over arrays of parameterized length. In our framework we consider properties of parameterized protocols that can be expressed by using formulas of the CTL temporal logic. Then, we show how these properties can be encoded by stratified logic programs with array formulas. Finally, we show how, in our general approach to protocol verification, CTL properties can be proved by applying unfold/fold transformation rules to a given specification. In particular, to show the effectiveness of this method, we prove that the parameterized Peterson's protocol enforces mutually exclusive access to a resource, for any given number of concurrent processes. The results illustrated in this chapter have been published in [58].

In Appendix A we present a run of the implementation of our method for the automatic generation of correctness proofs for program transformations discussed in Chapter 2. This demonstration has been done using the MAP transformations system. The proposed examples are taken from the literature. In particular, in the first example taken from [31] we show the

derivation of a recursive definition for the predicate *In_Correct_Position*. In the second example taken from [33] we obtain a more efficient definition for the predicate *Adjacent* which checks whether or not two elements have adjacent occurrences in a list. Finally, in the third example taken from [66] we prove a liveness property of an n-bit shift register whose specification is given as a logic program. In this example we also show how to prove the correctness of a goal replacement transformation step by using our method presented in Chapter 2.

In Appendix B we provide the definitions of some array formulas constructed during the verification of the parameterized Peterson's protocol, discussed in Chapter 5.

Chapter 2

Automatic Correctness Proofs for Logic Program Transformations

Rule-based program transformation is a program development methodology which consists in deriving from an initial program a final program, via the application of semantics preserving transformation rules [17]. In the field of logic (or functional) programming, program transformation can be regarded as a deductive process. Indeed, programs are logical (or equational, resp.) theories and the transformation rules can be viewed as rules for deducing new formulas from old ones. The logical soundness of the transformation rules easily implies that a transformation is *partially correct*, which means that an atom (or an equation, respectively) is true in the final program only if it is true in the initial program. However, it is usually much harder to prove that a transformation is *totally correct*, which means that an atom (or an equation, respectively) is true in the initial program if and only if it is true in the final program.

In the context of functional programming, it has been pointed out in the seminal paper by Burstall and Darlington [17] that, if the transformation rules rewrite the equations of the program at hand by using equations

which belong to the same program (like the *folding* and *unfolding* rules), the transformations are always partially correct, but the final program may terminate (w.r.t. a suitable notion of termination) less often than the initial one. Thus, a sufficient condition for total correctness is that the final program obtained by transformation always terminates. This method of proving total correctness is sometimes referred to as *McCarthy's method* [47]. However, the termination condition may be, in practice, very hard to check.

The situation is similar in the case of definite logic programs, where the folding and unfolding rules basically consist in applying equivalences that hold in the least Herbrand model of the initial program. For instance, let us consider the program:

P: $\quad p \leftarrow q \qquad\qquad r \leftarrow q \qquad\qquad q \leftarrow$

The least Herbrand model of P is $M(P) = \{p, q, r\}$ and $M(P) \models p \leftrightarrow q$. If we replace q by p in $r \leftarrow q$ (that is, we fold $r \leftarrow q$ using $p \leftarrow q$), then we get:

Q: $\quad p \leftarrow q \qquad\qquad r \leftarrow p \qquad\qquad q \leftarrow$

The transformation of P into Q is totally correct, because $M(P) = M(Q)$. However, if we replace q by p in $p \leftarrow q$ (that is, we fold $p \leftarrow q$ using $p \leftarrow q$ itself), then we get:

R: $\quad p \leftarrow p \qquad\qquad r \leftarrow q \qquad\qquad q \leftarrow$

and the transformation of P into R is partially correct, because $M(P) \supseteq M(R)$, but it is *not* totally correct, because $M(P) \neq M(R)$. Indeed, program R does not terminate for the goal p.

A lot of work has been devoted to devise methods for proving the total correctness of transformations based on various sets of rules, including the folding and the unfolding rules. These methods have been proposed both in the context of functional programming (see, for instance, [17, 35, 70]) and in the context of logic programming (see, for instance, [12, 13, 19, 24, 31, 33, 38, 56, 65, 66, 73, 79, 80]).

Some of these methods (such as, [12, 17, 19, 38]) propose sufficient conditions for total correctness which are explicitly based on the preservation of suitable termination properties (such as, termination of call-by-name re-

duction for functional programs, and universal or existential termination for logic programs).

Other methods, which we may call *implicit methods*, are based on conditions on the sequence of applications of the transformation rules that guarantee that termination is preserved. A notable example of these implicit methods is presented in [33], where integer counters are associated with program clauses. The counters of the initial program are set to 1 and are incremented (or decremented) when an unfolding (or folding, respectively) is applied. A sequence of transformations is totally correct if the counters of the clauses of the final program are all positive.

The method based on counters allows us to prove the total correctness of many transformations. Unfortunately, there are also many simple derivations where the method fails to guarantee the total correctness. For instance, in the transformation from P to Q described above, we would get a value of 0 for the counter of the clause $r \leftarrow p$ in the final program Q, because it has been derived by applying the folding rule from clause $r \leftarrow p$. Thus, the method does not yield the total correctness of the transformation. In order to overcome the limitations of the basic counter method, some modifications and enhancements have been described in [33, 65, 66, 80], where each clause is given a *measure* which is more complex than an integer counter.

In this chapter we present a different approach to the improvement of the basic counter method: instead of fixing *in advance* complex clause measures, for any given transformation we automatically generate, if at all possible, the clause measures that prove its correctness. For reasons of simplicity we assume that clause measures are non-negative integers, also called *weights*, and given a transformation starting from a program P, we look for a weight assignment to the clauses of P that proves that the transformation is totally correct.

This chapter is structured as follows. In Section 2.1 we present the notion of a *weighted transformation sequence*, that is, a sequence of programs constructed by applying suitable variants of the definition introduction, unfolding, and folding rules. We associate the clauses of the initial program of the sequence with some unknown weights, and during the construction of

the sequence, we generate a set of constraints consisting of linear equations and inequations which relate those weights. If the final set of constraints is satisfiable for some assignment to the unknown weights, then the transformation sequence is totally correct. In Section 2.2 we prove our total correctness result which is based on the *well-founded annotations* method proposed in [56]. In Section 2.3 we consider transformation sequences constructed by using also the goal replacement rule and we present a method for proving the total correctness of those transformation sequences. Finally, in Section 2.4 we present a method for proving predicate properties which are needed for applying the goal replacement rule.

2.1 Weighted Unfold/Fold Transformation Rules

Let us begin by introducing some terminology concerning systems of linear equations and inequations with integer coefficients and non-negative integer solutions.

By $\mathcal{P}_{LIN}$ we denote the set of linear polynomials with integer coefficients. Variables occurring in polynomials are called *unknowns* to distinguish them from logical variables occurring in programs. By $\mathcal{C}_{LIN}$ we denote the set of linear equations and inequations with integer coefficients, that is, $\mathcal{C}_{LIN}$ is the set $\{p_1 = p_2,\ p_1 < p_2,\ p_1 \leq p_2 \mid p_1, p_2 \in \mathcal{P}_{LIN}\}$. By $p_1 \geq p_2$ we mean $p_2 \leq p_1$, and by $p_1 > p_2$ we mean $p_2 < p_1$. An element of $\mathcal{C}_{LIN}$ is called a *constraint*. A *valuation* for a set $\{u_1, \ldots, u_r\}$ of unknowns is a mapping $\sigma : \{u_1, \ldots, u_r\} \to \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers. Let $\{u_1, \ldots, u_r\}$ be the set of unknowns occurring in $p \in \mathcal{P}_{LIN}$. Given a valuation σ for (a superset of) $\{u_1, \ldots, u_r\}$, $\sigma(p)$ is the integer obtained by replacing the occurrences of $u_1, \ldots, u_r$ in p by $\sigma(u_1), \ldots, \sigma(u_r)$, respectively, and then computing the value of the resulting arithmetic expression. A valuation σ is a *solution* for the constraint $p_1 = p_2$ if σ is a valuation for a superset of the variables occurring in $p_1 = p_2$ and $\sigma(p_1) = \sigma(p_2)$ holds. Similarly, we define a solution for $p_1 < p_2$ and for $p_1 \leq p_2$. σ is a solution for a finite set $\mathcal{C}$ of constraints if, for every $c \in \mathcal{C}$, σ is a solution for c. We say that a constraint c is *satisfiable* if there exists a solution for c. Similarly,

we say that a set $\mathcal{C}$ of constraints is *satisfiable* if there exists a solution for $\mathcal{C}$. A *weight function* for a set S of clauses is a function $\gamma : S \to \mathcal{P}_{LIN}$. A value of γ is also called a *weight polynomial*.

A *weighted unfold/fold transformation sequence* is a sequence of programs, denoted $P_0 \mapsto P_1 \mapsto \cdots \mapsto P_n$, such that $n \geq 0$ and, for $k = 0, \ldots, n-1$, P_{k+1} is derived from P_k by applying one of the following transformation rules: *weighted definition introduction*, *weighted unfolding*, and *weighted folding*. These rules, which will be defined below, are variants of the familiar rules without weights. For reasons of simplicity, when referring to the transformation rules, we will often omit the qualification 'weighted'. For $k = 0, \ldots, n$, we will define: (i) a weight function $\gamma_k : P_k \to \mathcal{P}_{LIN}$, (ii) a finite set $\mathcal{C}_k$ of constraints, (iii) a set $Defs_k$ of clauses defining the new predicates introduced by the definition introduction rule during the construction of the sequence $P_0 \mapsto P_1 \mapsto \cdots \mapsto P_k$, and (iv) a weight function $\delta_k : P_0 \cup Defs_k \to \mathcal{P}_{LIN}$. The weight function γ_0 for the initial program P_0 is defined as follows: for every clause $C \in P_0$, $\gamma_0(C) = u$, where u is an unknown and, for each pair C and D of distinct clauses in P_0, we have that $\gamma_0(C) \neq \gamma_0(D)$. The initial sets $\mathcal{C}_0$ and $Defs_0$ are, by definition, equal to the empty set and $\delta_0 = \gamma_0$.

For every $k > 0$, we assume that P_0 and P_k have no variables in common. This assumption is not restrictive because we can always rename the variables occurring in a program without affecting its least Herbrand model. Indeed, in the sequel we will feel free to rename variables, whenever needed.

Rule 1 (Weighted Definition Introduction) Let $D_1, \ldots, D_m$, with $m > 0$, be clauses such that, for $i = 1, \ldots, m$, the predicate of the head of D_i does not occur in $P_0 \cup Defs_k$. By *definition introduction* from P_k we derive $P_{k+1} = P_k \cup \{D_1, \ldots, D_m\}$.
We set the following: (1.1) for all C in P_k, $\gamma_{k+1}(C) = \gamma_k(C)$, (1.2) for $i = 1, \ldots, m$, $\gamma_{k+1}(D_i) = u_i$, where u_i is a fresh new unknown, (2) $\mathcal{C}_{k+1} = \mathcal{C}_k$, (3) $Defs_{k+1} = Defs_k \cup \{D_1, \ldots, D_m\}$, (4.1) for all D in $P_0 \cup Defs_k$, $\delta_{k+1}(D) = \delta_k(D)$, and (4.2) for $i = 1, \ldots, m$, $\delta_{k+1}(D_i) = u_i$.

Rule 2 (Weighted Unfolding) Let C: $H \leftarrow G_L \wedge A \wedge G_R$ be a clause in P_k and let C_1: $H_1 \leftarrow G_1$, ..., C_m: $H_m \leftarrow G_m$, with $m \geq 0$, be *all* clauses

in P_0 such that, for $i = 1, \ldots, m$, A is unifiable with H_i via a most general unifier ϑ_i. By *unfolding* C w.r.t. A *using* $C_1, \ldots, C_m$, we derive the clauses D_1: $(H \leftarrow G_L \wedge G_1 \wedge G_R)\vartheta_1$, ..., D_m: $(H \leftarrow G_L \wedge G_m \wedge G_R)\vartheta_m$, and from P_k we derive $P_{k+1} = (P_k - \{C\}) \cup \{D_1, \ldots, D_m\}$.
We set the following: (1.1) for all D in $P_k - \{C\}$, $\gamma_{k+1}(D) = \gamma_k(D)$, (1.2) for $i = 1, \ldots, m$, $\gamma_{k+1}(D_i) = \gamma_k(C) + \gamma_0(C_i)$, (2) $\mathcal{C}_{k+1} = \mathcal{C}_k$, (3) $Defs_{k+1} = Defs_k$, and (4) $\delta_{k+1} = \delta_k$.

For a goal (or set of goals) G, by $vars(G)$ we denote the set of variables occurring in G.

Rule 3 (Weighted Folding) Let C_1: $H \leftarrow G_L \wedge G_1 \wedge G_R$, ..., C_m: $H \leftarrow G_L \wedge G_m \wedge G_R$ be clauses in P_k and let D_1: $K \leftarrow B_1$, ..., D_m: $K \leftarrow B_m$ be clauses in $P_0 \cup Defs_k$. Suppose that there exists a substitution ϑ such that the following conditions hold: (i) for $i = 1, \ldots, m$, $G_i = B_i\vartheta$, (ii) there exists no clause in $(P_0 \cup Defs_k) - \{D_1, \ldots, D_m\}$ whose head is unifiable with $K\vartheta$, and (iii) for $i = 1, \ldots, m$ and for every variable U in $vars(B_i) - vars(K)$: (iii.1) $U\vartheta$ is a variable not occurring in $\{H, G_L, G_R\}$, and (iii.2) $U\vartheta$ does not occur in the term $V\vartheta$, for any variable V occurring in B_i and different from U.
By *folding* $C_1, \ldots, C_m$ *using* $D_1, \ldots, D_m$, we derive E: $H \leftarrow G_L \wedge K\vartheta \wedge G_R$, and from P_k we derive $P_{k+1} = (P_k - \{C_1, \ldots, C_m\}) \cup \{E\}$.
We set the following: (1.1) for all C in $P_k - \{C_1, \ldots, C_m\}$, $\gamma_{k+1}(C) = \gamma_k(C)$, (1.2) $\gamma_{k+1}(E) = u$, where u is a new unknown, (2) $\mathcal{C}_{k+1} = \mathcal{C}_k \cup \{u \leq \gamma_k(C_1) - \delta_k(D_1), \ldots, u \leq \gamma_k(C_m) - \delta_k(D_m)\}$, (3) $Defs_{k+1} = Defs_k$, and (4) $\delta_{k+1} = \delta_k$.

The *correctness constraint system* associated with a weighted unfold/fold transformation sequence $P_0 \mapsto \cdots \mapsto P_n$ is the set $\mathcal{C}_{final}$ of constraints defined as follows:

$$\mathcal{C}_{final} = \mathcal{C}_n \cup \{\gamma_n(C) \geq 1 \mid C \in P_n\}.$$

The following result, which will be proved in Section 2.2, guarantees the total correctness of weighted unfold/fold transformations. By $M(P)$ we denote the least Herbrand model of program P.

Theorem 1 (Total Correctness of Weighted Unfold/Fold Transformations) *Let $P_0 \mapsto \cdots \mapsto P_n$ be a weighted unfold/fold transformation sequence constructed by using Rules* 1–3*, and let $\mathcal{C}_{final}$ be its associated correctness constraint system. If $\mathcal{C}_{final}$ is satisfiable then $M(P_0 \cup Defs_n) = M(P_n)$.*

Now we give an example of application of the weighted unfold/fold transformation rules.

Example 1 (*Continuation Passing Style Transformation*) Let us consider the initial program P_0 consisting of the following three clauses whose weight polynomials are the unknowns u_1, u_2, and u_3, respectively (we write weight polynomials on a second column to the right of the corresponding clause):

1. $p \leftarrow$ u_1
2. $p \leftarrow p \wedge q$ u_2
3. $q \leftarrow$ u_3

We want to derive a continuation-passing-style program defining a predicate p_{cont} equivalent to the predicate p defined by the program P_0. In order to do so, we introduce by Rule 1 the following clause 4 with its unknown u_4:

4. $p_{cont} \leftarrow p$ u_4

and also the following three clauses for the unary continuation predicate *cont* with unknowns u_5, u_6, and u_7, respectively:

$(*)$5. $cont(f_{true}) \leftarrow$ u_5
6. $cont(f_p(X)) \leftarrow p \wedge cont(X)$ u_6
7. $cont(f_q(X)) \leftarrow q \wedge cont(X)$ u_7

where f_{true}, f_p, and f_q are three function symbols corresponding to the three predicates *true*, p, and q, respectively. By folding clause 4 using clause 5 we get the following clause with the unknown u_8 which should satisfy the constraint $u_8 \leq u_4 - u_5$ (we write constraints on a third column to the right of the corresponding clause):

8. $p_{cont} \leftarrow p \wedge cont(f_{true})$ u_8 $u_8 \leq u_4 - u_5$

By folding clause 8 using clause 6 we get the following clause 9 with unknown u_9 such that $u_9 \leq u_8 - u_6$:

$(*)9.\ \ p_{cont} \leftarrow cont(f_p(f_{true}))$ $\qquad u_9$ $\qquad u_9 \leq u_8 - u_6$

By unfolding clause 6 w.r.t. p using clauses 1 and 2, we get:

$(*)10.\ \ cont(f_p(X)) \leftarrow cont(X)$ $\qquad u_6 + u_1$

$11.\ \ cont(f_p(X)) \leftarrow p \wedge q \wedge cont(X)$ $\qquad u_6 + u_2$

Then by folding clause 11 using clause 7 we get:

$12.\ \ cont(f_p(X)) \leftarrow p \wedge cont(f_q(X))$ $\qquad u_{12}$ $\qquad u_{12} \leq u_6 + u_2 - u_7$

and by folding clause 12 using clause 6 we get:

$(*)13.\ \ cont(f_p(X)) \leftarrow cont(f_p(f_q(X)))$ $\qquad u_{13}$ $\qquad u_{13} \leq u_{12} - u_6$

Finally, by unfolding clause 7 w.r.t. q we get:

$(*)14.\ \ cont(f_q(X)) \leftarrow cont(X)$ $\qquad u_7 + u_3$

The final program is made out of clauses 5, 9, 10, 13, and 14, marked with $(*)$, and clauses 1, 2, and 3. The correctness constraint system $\mathcal{C}_{final}$ is made out of the following 11 constraints.

For clauses 5, 9, 10, 13, and 14:

$$u_5 \geq 1,\ \ u_9 \geq 1,\ \ u_6 + u_1 \geq 1,\ \ u_{13} \geq 1,\ \ u_7 + u_3 \geq 1.$$

For clauses 1, 2, and 3:

$$u_1 \geq 1,\ \ u_2 \geq 1,\ \ u_3 \geq 1.$$

For the four folding steps:

$$u_8 \leq u_4 - u_5,\ \ u_9 \leq u_8 - u_6,\ \ u_{12} \leq u_6 + u_2 - u_7,\ \ u_{13} \leq u_{12} - u_6$$

This system $\mathcal{C}_{final}$ of constraints is satisfiable and thus, the transformation from program P_0 to the final program is totally correct. □

2.2 Proving Correctness Via Weighted Programs

In order to prove that a weighted unfold/fold transformation sequence $P_0 \mapsto \cdots \mapsto P_n$ is *totally correct* (see Theorem 1), we specialize the method based on *well-founded annotations* proposed in [56]. In particular, with each program P_k in the transformation sequence, we associate a *weighted program* $\overline{P}_k$ by adding an integer argument $n(\geq 0)$, called a *weight*, to each atom $p(\vec{t})$ occurring in P_k. Here and in the sequel, $\vec{t}$ denotes a generic

m-tuple of terms $t_1, \ldots, t_m$, for some $m \geq 0$. Informally, $p(\vec{t}, n)$ holds in $\overline{P}_k$ if $p(\vec{t})$ 'has a proof of weight at least n' in P_k. We will show that if the correctness constraint system $\mathcal{C}_{final}$ is satisfiable, then it is possible to derive from $\overline{P}_0$ a weighted program $\overline{P}_n$ where the weight arguments determine, for every clause $\overline{C}$ in $\overline{P}_n$, a well-founded ordering between the head of $\overline{C}$ and every atom in the body $\overline{C}$. Hence $\overline{P}_n$ terminates for all ground goals (even if P_n need not) and the immediate consequence operator $T_{\overline{P}}$ has a unique fixpoint [10]. Thus, as proved in [56], the total correctness of the transformation sequence follows from the *unique fixpoint principle* (see Corollary 1).

Our transformation rules can be regarded as rules for replacing a set of clauses by an equivalent one. Let us introduce the notions of implication and equivalence between sets of clauses according to [56].

Definition 1 Let I be an Herbrand interpretation and let Γ_1 and Γ_2 be two sets of clauses. We write $I \models \Gamma_1 \Rightarrow \Gamma_2$ if for every ground instance $H \leftarrow G_2$ of a clause in Γ_2 such that $I \models G_2$ there exists a ground instance $H \leftarrow G_1$ of a clause in Γ_1 such that $I \models G_1$. We write $I \models \Gamma_1 \Leftarrow \Gamma_2$ if $I \models \Gamma_2 \Rightarrow \Gamma_1$, and we write $I \models \Gamma_1 \Leftrightarrow \Gamma_2$ if ($I \models \Gamma_1 \Rightarrow \Gamma_2$ and $I \models \Gamma_1 \Leftarrow \Gamma_2$).

For all Herbrand interpretations I and sets of clauses Γ_1, Γ_2, and Γ_3 the following properties hold:

Reflexivity: $I \models \Gamma_1 \Rightarrow \Gamma_1$
Transitivity: if $I \models \Gamma_1 \Rightarrow \Gamma_2$ and $I \models \Gamma_2 \Rightarrow \Gamma_3$ then $I \models \Gamma_1 \Rightarrow \Gamma_3$
Monotonicity: if $I \models \Gamma_1 \Rightarrow \Gamma_2$ then $I \models \Gamma_1 \cup \Gamma_3 \Rightarrow \Gamma_2 \cup \Gamma_3$.

Given a program P, we denote its associated *immediate consequence operator* by T_P [1, 43]. We denote the least and greatest fixpoint of T_P by $lfp(T_P)$ and $gfp(T_P)$, respectively. Recall that $M(P) = lfp(T_P)$.

Now let us consider the transformation of a program P into a program Q consisting in the replacement of a set Γ_1 of clauses in P by a new set Γ_2 of clauses. The following result, proved in [56], expresses the *partial correctness* of the transformation of P into Q.

Theorem 2 (Partial Correctness) *Given two programs P and Q, such that*: (i) *for some sets* Γ_1 *and* Γ_2 *of clauses,* $Q = (P - \Gamma_1) \cup \Gamma_2$, *and* (ii) $M(P) \models \Gamma_1 \Rightarrow \Gamma_2$. *Then* $M(P) \supseteq M(Q)$.

In order to establish a sufficient condition for the total correctness of the transformation of P into Q, that is, $M(P) = M(Q)$, we consider programs whose associated immediate consequence operators have unique fixpoints.

Definition 2 (Univocal Program) A program P is said to be *univocal* if T_P has a unique fixpoint, that is, $lfp(T_P) = gfp(T_P)$.

The following theorem is proved in [56].

Theorem 3 (Conservativity) *Given two programs P and Q, such that*: (i) *for some sets* Γ_1 *and* Γ_2 *of clauses,* $Q = (P - \Gamma_1) \cup \Gamma_2$, (ii) $M(P) \models \Gamma_1 \Leftarrow \Gamma_2$, *and* (iii) Q *is univocal. Then* $M(P) \subseteq M(Q)$.

As a straightforward consequence of Theorems 2 and 3 we get the following.

Corollary 1 (Total Correctness Via Unique Fixpoint) *Given two programs P and Q such that*: (i) *for some sets* Γ_1, Γ_2 *of clauses,* $Q = (P - \Gamma_1) \cup \Gamma_2$, (ii) $M(P) \models \Gamma_1 \Leftrightarrow \Gamma_2$, *and* (iii) Q *is univocal. Then* $M(P) = M(Q)$.

Corollary 1 cannot be directly applied to prove the total correctness of a transformation sequence generated by applying the unfolding and folding rules, because the programs derived by these rules need not be univocal. To overcome this difficulty we introduce the notion of weighted program.

Given a clause C of the form $p_0(\vec{t_0}) \leftarrow p_1(\vec{t_1}) \wedge \ldots \wedge p_m(\vec{t_m})$, where $\vec{t_0}, \vec{t_1}, \ldots, \vec{t_m}$ are tuples of terms, a *weighted clause*, denoted $\overline{C}(w)$, associated with C is a clause of the form:

$\overline{C}(w)$: $p_0(\vec{t_0}, N_0) \leftarrow N_0 \geq N_1 + \cdots + N_m + w \wedge p_1(\vec{t_1}, N_1) \wedge \ldots \wedge p_m(\vec{t_m}, N_m)$

where w is a natural number called the *weight* of $\overline{C}(w)$. Clause $\overline{C}(w)$ is denoted by $\overline{C}$ when we do not need refer to the weight w. A *weighted program*

is a set of weighted clauses. Given a program $P = \{C_1, \ldots, C_r\}$, by $\overline{P}$ we denote a weighted program of the form $\{\overline{C}_1, \ldots, \overline{C}_r\}$. Given a weight function γ and a valuation σ, by $\overline{C}(\gamma, \sigma)$ we denote the weighted clause $\overline{C}(\sigma(\gamma(C)))$ and by $\overline{P}(\gamma, \sigma)$ we denote the weighted program $\{\overline{C}_1(\gamma, \sigma), \ldots, \overline{C}_r(\gamma, \sigma)\}$.

For reasons of conciseness, we do not formally define here when a formula of the form $N_0 \geq N_1 + \cdots + N_m + w$ (see clause $\overline{C}(w)$ above) holds in an interpretation, and we simply say that for every Herbrand interpretation I and ground terms $n_0, n_1, \ldots, n_m, w$, we have that $I \models n_0 \geq n_1 + \cdots + n_m + w$ holds iff $n_0, n_1, \ldots, n_m, w$ are (terms representing) natural numbers such that n_0 is greater than or equal to $n_1 + \cdots + n_m + w$.

The following lemma (proved in [56]) establishes the relationship between the semantics of a program P and the semantics of any weighted program $\overline{P}$ associated with P.

Lemma 1 *Let P be a program. For every ground atom $p(\vec{t})$, $p(\vec{t}) \in M(P)$ iff there exists $n \in \mathbb{N}$ such that $p(\vec{t}, n) \in M(\overline{P})$.*

By erasing weights from clauses we preserve clause implications, in the sense stated by the following lemma (proved in [56]).

Lemma 2 *Let P be a program, and Γ_1 and Γ_2 be any two sets of clauses. If $M(\overline{P}) \models \overline{\Gamma}_1 \Rightarrow \overline{\Gamma}_2$ then $M(P) \models \Gamma_1 \Rightarrow \Gamma_2$.*

A weighted program $\overline{P}$ is said to be *decreasing* if every clause in $\overline{P}$ has a positive weight.

Lemma 3 *Every decreasing program is univocal.*

Now, we have the following result, which is a consequence of Lemmata 1, 3, and Theorems 2 and 3. Unlike Corollary 1, this result can be used to prove the total correctness of the transformation of program P into program Q also in the case where Q is not univocal.

Theorem 4 (Total Correctness Via Weights) *Let P and Q be programs such that: (i) $M(P) \models P \Rightarrow Q$, (ii) $M(\overline{P}) \models \overline{P} \Leftarrow \overline{Q}$, and (iii) $\overline{Q}$ is decreasing. Then $M(P) = M(Q)$.*

By Theorem 4, in order to prove Theorem 1, that is, the total correctness of weighted unfold/fold transformations, it is enough to show that, given a weighted unfold/fold transformation sequence $P_0 \mapsto \cdots \mapsto P_n$, we have that:

(P1) $M(P_0 \cup \mathit{Defs}_n) \models P_0 \cup \mathit{Defs}_n \Rightarrow P_n$

and there exist suitable weighted programs $\overline{P}_0 \cup \overline{\mathit{Defs}}_n$ and $\overline{P}_n$, associated with $P_0 \cup \mathit{Defs}_n$ and P_n, respectively, such that:

(P2) $M(\overline{P}_0 \cup \overline{\mathit{Defs}}_n) \models \overline{P}_0 \cup \overline{\mathit{Defs}}_n \Leftarrow \overline{P}_n$, and

(P3) $\overline{P}_n$ is decreasing.

The suitable weighted programs $\overline{P}_0 \cup \overline{\mathit{Defs}}_n$ and $\overline{P}_n$ are constructed as we now indicate by using the hypothesis that the correctness constraint system $\mathcal{C}_{\mathit{final}}$ associated with the transformation sequence, is satisfiable. Let σ be a solution for $\mathcal{C}_{\mathit{final}}$. For every $k = 0, \ldots, n$ and for every clause $C \in P_k$, we take $\overline{C} = \overline{C}(\gamma_k, \sigma)$, where γ_k is the weight function associated with P_k. For $C \in \mathit{Defs}_k$ we take $\overline{C} = \overline{C}(\delta_k, \sigma)$. Thus, $\overline{P}_k = \overline{P}_k(\gamma_k, \sigma)$ and $\overline{\mathit{Defs}}_k = \overline{\mathit{Defs}}_k(\delta_k, \sigma)$.

In order to prove Theorem 1 we need the following two lemmata.

Lemma 4 *Let $P_0 \mapsto \cdots \mapsto P_k$ be a weighted unfold/fold transformation sequence. Let C be a clause in P_k, and let $D_1, \ldots, D_m$ be the clauses derived by* unfolding *C w.r.t. an atom in its body, as described in Rule* 2. *Then*:

$M(\overline{P}_0 \cup \overline{\mathit{Defs}}_n) \models \{\overline{C}\} \Leftrightarrow \{\overline{D}_1, \ldots, \overline{D}_m\}$

Lemma 5 *Let $P_0 \mapsto \cdots \mapsto P_k$ be a weighted unfold/fold transformation sequence. Let $C_1, \ldots, C_m$ be clauses in P_k, $D_1, \ldots, D_m$ be clauses in $P_0 \cup \mathit{Defs}_k$, and E be the clause derived by* folding *$C_1, \ldots, C_m$ using $D_1, \ldots, D_m$, as described in Rule* 3. *Then*:

(*i*) $M(P_0 \cup \mathit{Defs}_n) \models \{C_1, \ldots, C_m\} \Rightarrow \{E\}$

(*ii*) $M(\overline{P}_0 \cup \overline{\mathit{Defs}}_n) \models \{\overline{C}_1, \ldots, \overline{C}_m\} \Leftarrow \{\overline{E}\}$

We are now able to prove Theorem 1. For a weighted unfold/fold transformation sequence $P_0 \mapsto \cdots \mapsto P_n$, the following properties hold:

(R1) $M(P_0 \cup Defs_n) \models P_k \cup (Defs_n - Defs_k) \Rightarrow P_{k+1} \cup (Defs_n - Defs_{k+1})$, and

(R2) $M(\overline{P}_0 \cup \overline{Defs}_n) \models \overline{P}_k \cup (\overline{Defs}_n - \overline{Defs}_k) \Leftarrow \overline{P}_{k+1} \cup (\overline{Defs}_n - \overline{Defs}_{k+1})$.

Indeed, Properties (R1) and (R2) can be proved by reasoning by cases on the transformation rule applied to derive P_{k+1} from P_k, as follows. If P_{k+1} is derived from P_k by applying the definition introduction rule then $P_k \cup (Defs_n - Defs_k) = P_{k+1} \cup (Defs_n - Defs_{k+1})$ and, therefore, Properties (R1) and (R2) are trivially true. If P_{k+1} is derived from P_k by applying the unfolding rule, then $P_{k+1} = (P_k - \{C\}) \cup \{D_1, \ldots, D_m\}$ and $Defs_k = Defs_{k+1}$. Hence, Properties (R1) and (R2) follow from Lemma 2, Lemma 4 and from the monotonicity of $\Rightarrow$. If P_{k+1} is derived from P_k by applying the folding rule, then $P_{k+1} = (P_k - \{C_1, \ldots, C_m\}) \cup \{E\}$ and $Defs_k = Defs_{k+1}$. Hence, Properties (R1) and (R2) follow from Points (i) and (ii) of Lemma 5 and the monotonicity of $\Rightarrow$.

By the transitivity of $\Rightarrow$ and by Properties (R1) and (R2), we get Properties (P1) and (P2). Moreover, since σ is a solution for $\mathcal{C}_{final}$ and $\overline{P}_n = \overline{P}_n(\gamma_n, \sigma)$, Property (P3) holds. Thus, by Theorem 4, $M(P_0 \cup Defs_n) = M(P_n)$.

2.3 Weighted Goal Replacement

In this section we extend the notion of a weighted unfold/fold transformation sequence $P_0 \mapsto P_1 \mapsto \cdots \mapsto P_n$ by assuming that P_{k+1} is derived from P_k by applying, besides the definition introduction, unfolding, and folding rules, also the goal replacement rule defined as Rule 4 below. The goal replacement rule consists in replacing a goal G_1 occurring in the body of a clause of P_k, by a new goal G_2 such that G_1 and G_2 are equivalent in $M(P_0 \cup Defs_k)$. Some conditions are also needed in order to update the value of the weight function and the associated constraints. To define these conditions we introduce the notion of *weighted replacement law* (see Definition 3), which in turn is based on the notion of weighted program introduced in Section 2.2.

In Definition 3 below we will use the following notation. Given a goal

$G : p_1(\vec{t_1}) \wedge \ldots \wedge p_m(\vec{t_m})$, a variable N, and a natural number w, by $\overline{G}[N, w]$ we denote the formula $\exists N_1 \ldots \exists N_m (N \geq N_1 + \cdots + N_m + w \wedge p_1(\vec{t_1}, N_1) \wedge \ldots \wedge p_m(\vec{t_m}, N_m))$. Given a set $X = \{X_1, \ldots, X_m\}$ of variables, we will use '$\exists X$' as a shorthand for '$\exists X_1 \ldots \exists X_m$' and '$\forall X$' as a shorthand for '$\forall X_1 \ldots \forall X_m$'.

Definition 3 (Weighted Replacement Law) Let P be a program, γ be a weight function for P, and $\mathcal{C}$ be a finite set of constraints. Let G_1 and G_2 be goals, u_1 and u_2 be unknowns, and $X \subseteq \mathit{vars}(G_1) \cup \mathit{vars}(G_2)$ be a set of variables. We say that the *weighted replacement law* $(G_1, u_1) \Rightarrow_X (G_2, u_2)$ holds with respect to the triple $\langle P, \gamma, \mathcal{C} \rangle$, and we write $\langle P, \gamma, \mathcal{C} \rangle \models (G_1, u_1) \Rightarrow_X (G_2, u_2)$, if the following conditions hold:

(i) $M(P) \models \forall X \, (\exists Y \, G_1 \leftarrow \exists Z \, G_2)$, and

(ii) for every solution σ for $\mathcal{C}$,

$M(\overline{P}) \models \forall X \forall U \, (\exists Y \, (\overline{G}_1[U, \sigma(u_1)]) \rightarrow \exists Z \, (\overline{G}_2[U, \sigma(u_2)]))$

where: (1) $\overline{P}$ is the weighted program $\overline{P}(\gamma, \sigma)$, (2) U is a variable, (3) $Y = \mathit{vars}(G_1) - X$, and (4) $Z = \mathit{vars}(G_2) - X$.

By using Lemma 2 it can be shown that, if $\mathcal{C}$ is satisfiable, then Condition (ii) of Definition 3 implies $M(P) \models \forall X \, (\exists Y \, G_1 \rightarrow \exists Z \, G_2)$ and, therefore, if $\langle P, \gamma, \mathcal{C} \rangle \models (G_1, u_1) \Rightarrow_X (G_2, u_2)$ and $\mathcal{C}$ is satisfiable, we have that $M(P) \models \forall X \, (\exists Y \, G_1 \leftrightarrow \exists Z \, G_2)$.

Example 2 (*Associativity of List Concatenation*) Let us consider the following program *Append* for list concatenation. To the right of each clause we indicate the corresponding unknown.

1. $a([\,], L, L)$ $\qquad u_1$
2. $a([H|T], L, [H|R]) \leftarrow a(T, L, R)$ $\qquad u_2$

The following replacement law expresses the associativity of list concatenation:

Law (α): $(a(L_1, L_2, M) \wedge a(M, L_3, L), \; w_1)$
$\Rightarrow_{\{L_1, L_2, L_3, L\}} \quad (a(L_2, L_3, R) \wedge a(L_1, R, L), \; w_2)$

where w_1 and w_2 are new unknowns. In Example 4 below we will show that Law (α) holds w.r.t. $\langle \mathit{Append}, \gamma, \mathcal{C}\rangle$, where $\mathcal{C}$ is the set of constraints $\{u_1 \geq 1,\ u_2 \geq 1,\ w_1 \geq w_2,\ w_2 + u_1 \geq 1\}$. □

In Section 2.4 we will present a method, called *weighted unfold/fold proof method*, for generating a suitable set $\mathcal{C}$ of constraints such that $\langle P, \gamma, \mathcal{C}\rangle \models (G_1, u_1) \Rightarrow_X (G_2, u_2)$ holds.

Now we introduce the Weighted Goal Replacement Rule, which is a variant of the rule without weights (see, for instance, [79]).

Rule 4 (Weighted Goal Replacement) Let C: $H \leftarrow G_L \wedge G_1 \wedge G_R$ be a clause in program P_k and let $\mathcal{C}$ be a set of constraints such that the weighted replacement law $\lambda : (G_1, u_1) \Rightarrow_X (G_2, u_2)$ holds w.r.t. $\langle P_0 \cup \mathit{Defs}_k, \delta_k, \mathcal{C}\rangle$, where $X = \mathit{vars}(\{H, G_L, G_R\}) \cap \mathit{vars}(\{G_1, G_2\})$.
By applying the replacement law λ, from C we derive $D : H \leftarrow G_L \wedge G_2 \wedge G_R$, and from P_k we derive $P_{k+1} = (P_k - \{C\}) \cup \{D\}$. We set the following: (1.1) for all E in $P_k - \{C\}$, $\gamma_{k+1}(E) = \gamma_k(E)$, (1.2) $\gamma_{k+1}(D) = \gamma_k(C) - u_1 + u_2$, (2) $\mathcal{C}_{k+1} = \mathcal{C}_k \cup \mathcal{C}$, (3) $\mathit{Defs}_{k+1} = \mathit{Defs}_k$, and (4) $\delta_{k+1} = \delta_k$.

The proof of the following result is similar to the one of Theorem 1.

Theorem 5 (Total Correctness of Weighted Unfold/Fold/Replacement Transformations) *Let $P_0 \mapsto \cdots \mapsto P_n$ be a weighted unfold/fold transformation sequence constructed by using Rules* 1–4, *and let $\mathcal{C}_{\mathit{final}}$ be its associated correctness constraint system. If $\mathcal{C}_{\mathit{final}}$ is satisfiable then $M(P_0 \cup \mathit{Defs}_n) = M(P_n)$.*

Next we give a simple example of a transformation sequence in which we apply the goal replacement rule.

Example 3 (*List Reversal*) Let *Reverse* be a program for list reversal consisting of the clauses of *Append* (see Example 2) together with the following

two clauses (to the right of the clauses we write the corresponding weight polynomials):

3. $r([\,],[\,]) \leftarrow$ u_3
4. $r([H|T],L) \leftarrow r(T,R) \wedge a(R,[H],L)$ u_4

We will transform the *Reverse* program into a program that uses an accumulator [17]. In order to do so, we introduce by Rule 1 the following clause:

5. $g(L_1,L_2,A) \leftarrow r(L_1,R) \wedge a(R,A,L_2)$ u_5

We apply the unfolding rule twice starting from clause 5 and we get:

6. $g([\,],L,L) \leftarrow$ $u_5+u_3+u_1$
7. $g([H|T],L,A) \leftarrow r(T,R) \wedge a(R,[H],S) \wedge a(S,A,L)$ u_5+u_4

By applying the replacement law (α), from clause 7 we derive:

8. $g([H|T],L,A) \leftarrow r(T,R) \wedge a([H],A,S) \wedge a(R,S,L)$ $u_5+u_4-w_1+w_2$

together with the constraints (see Example 2): $u_1 \geq 1,\ u_2 \geq 1,\ w_1 \geq w_2,\ w_2+u_1 \geq 1$.
By two applications of the unfolding rule, from clause 8 we get:

9. $g([H|T],L,A) \leftarrow r(T,R) \wedge a(R,[H|A],L)$ $u_5+u_4-w_1+w_2+u_2+u_1$

By folding clause 9 using clause 5 we get:

10. $g([H|T],L,A) \leftarrow g(T,L,[H|A])$ u_6

together with the constraint: $u_6 \leq u_4 - w_1 + w_2 + u_2 + u_1$.
Finally, by folding clause 4 using clause 5 we get:

11. $r([H|T],L) \leftarrow g(T,L,[H])$ u_7

together with the constraint: $u_7 \leq u_4 - u_5$.

The final program consists of clauses 1, 2, 3, 11, 6, and 10. The correctness constraint system associated with the transformation sequence is as follows.

For clauses 1, 2, and 3: $u_1 \geq 1,\ u_2 \geq 1,\ u_3 \geq 1$.
For clauses 11, 6, and 10: $u_7 \geq 1,\ u_5+u_3+u_1 \geq 1,\ u_6 \geq 1$.

For the goal replacement: $u_1 \geq 1,\ u_2 \geq 1,\ w_1 \geq w_2,\ w_2 + u_1 \geq 1$.
For the two folding steps: $u_6 \leq u_4 - w_1 + w_2 + u_2 + u_1,\ u_7 \leq u_4 - u_5$.

This set of constraints is satisfiable and, therefore, the transformation sequence is totally correct. □

2.4 The Weighted Unfold/Fold Proof Method

In this section we present the unfold/fold method for proving the replacement laws to be used in Rule 4. In order to do so, we introduce the notions of: (i) *syntactic equivalence*, (ii) *symmetric folding*, and (iii) *symmetric goal replacement*.

A *predicate renaming* is a bijective mapping $\rho : \mathit{Preds}_1 \to \mathit{Preds}_2$, where Preds_1 and Preds_2 are two sets of predicate symbols. Given a formula (or a set of formulas) F, by $\mathit{preds}(F)$ we denote the set of predicate symbols occurring in F. Suppose that $\mathit{preds}(F) \subseteq \mathit{Preds}_1$, then by $\rho(F)$ we denote the formula obtained from F by replacing every predicate symbol p by $\rho(p)$. Two programs Q and R are *syntactically equivalent* if there exists a predicate renaming $\rho : \mathit{preds}(Q) \to \mathit{preds}(R)$, such that $R = \rho(Q)$, modulo variable renaming.

An application of the folding rule by which from program P_k we derive program P_{k+1}, is said to be *symmetric* if $\mathcal{C}_{k+1}$ is set to $\mathcal{C}_k \cup \{u = \gamma_k(C_1) - \delta_k(D_1), \ldots, u = \gamma_k(C_m) - \delta_k(D_m)\}$ (see Point 2 of Rule 3).
Given a program P, a weight function γ, and a set $\mathcal{C}$ of constraints, we say that the replacement law $(G_1, u_1) \Rightarrow_X (G_2, u_2)$ holds *symmetrically* w.r.t. $\langle P, \gamma, \mathcal{C}\rangle$, and we write $\langle P, \gamma, \mathcal{C}\rangle \models (G_1, u_1) \Leftrightarrow_X (G_2, u_2)$, if the following condition holds:

(ii*) for every solution σ for $\mathcal{C}$,

$$M(\overline{P}) \models \forall X \forall U\, (\exists Y\, (\overline{G}_1[U, \sigma(u_1)]) \leftrightarrow \exists Z\, (\overline{G}_2[U, \sigma(u_2)]))$$

where $\overline{P}$, U, Y, and Z are defined as in Definition 3. Note that, by Lemma 2, Condition (ii*) implies Condition (i) of Definition 3. An application of the *goal replacement rule* is *symmetric* if it consists in applying a replacement law that holds symmetrically w.r.t. $\langle P_0 \cup \mathit{Defs}_k, \delta_k, \mathcal{C}\rangle$. A

weighted unfold/fold transformation sequence is said to be *symmetric* if it is constructed by applications of the definition and unfolding rules and by symmetric applications of the folding and goal replacement rules.

Now we are ready to present the weighted unfold/fold proof method, which is itself based on weighted unfold/fold transformations.

The Weighted Unfold/Fold Proof Method. Let us consider a program P, a weight function γ for P, and a replacement law $(G_1, u_1) \Rightarrow_X (G_2, u_2)$. Suppose that X is the set of variables $\{X_1, \ldots, X_m\}$ and let $\vec{X}$ denote the sequence $X_1, \ldots, X_m$.

Step 1. First we introduce two new predicates $new1$ and $new2$ defined by the following two clauses: D_1: $new1(\vec{X}) \leftarrow G_1$ and D_2: $new2(\vec{X}) \leftarrow G_2$, associated with the unknowns u_1 and u_2, respectively.

Step 2. Then we construct two weighted unfold/fold transformation sequences of the forms: $P \cup \{D_1\} \mapsto \cdots \mapsto Q$ and $P \cup \{D_2\} \mapsto \cdots \mapsto R$, such that the following three conditions hold:

(1) For $i = 1, 2$, the weight function associated with the initial program $P \cup \{D_i\}$ is γ_0^i defined as: $\gamma_0^i(C) = \gamma(C)$ if $C \in P$, and $\gamma_0^i(D_i) = u_i$;

(2) The final programs Q and R are syntactically equivalent; and

(3) The transformation sequence $P \cup \{D_2\} \mapsto \cdots \mapsto R$ is symmetric.

Step 3. Finally, we construct a set $\mathcal{C}$ of constraints as follows. Let γ_Q and γ_R be the weight functions associated with Q and R, respectively. Let $\mathcal{C}_Q$ and $\mathcal{C}_R$ be the correctness constraint systems associated with the transformation sequences $P \cup \{D_1\} \mapsto \cdots \mapsto Q$ and $P \cup \{D_2\} \mapsto \cdots \mapsto R$, respectively, and let ρ be the predicate renaming such that $\rho(Q) = R$. Suppose that both $\mathcal{C}_Q$ and $\mathcal{C}_R$ are satisfiable.

(3.1) Let the set $\mathcal{C}$ be $\{\gamma_Q(C) \geq \gamma_R(\rho(C)) \mid C \in Q\} \cup \mathcal{C}_Q \cup \mathcal{C}_R$. Then we infer:

$$\langle P, \gamma, \mathcal{C} \rangle \vdash_{UF} (G_1, u_1) \Rightarrow_X (G_2, u_2)$$

(3.2) Suppose that the transformation sequence $P \cup \{D_1\} \mapsto \cdots \mapsto Q$ is symmetric and let the set $\mathcal{C}$ be $\{\gamma_Q(C) = \gamma_R(\rho(C)) \mid C \in Q\} \cup \mathcal{C}_Q \cup \mathcal{C}_R$. Then we infer:

$\langle P, \gamma, \mathcal{C}\rangle \vdash_{UF} (G_1, u_1) \Leftrightarrow_X (G_2, u_2)$

It can be shown that the unfold/fold proof method is sound.

Theorem 6 (Soundness of the Unfold/Fold Proof Method)
If $\langle P, \gamma, \mathcal{C}\rangle \vdash_{UF} (G_1, u_1) \Rightarrow_X (G_2, u_2)$ *then* $\langle P, \gamma, \mathcal{C}\rangle \models (G_1, u_1) \Rightarrow_X (G_2, u_2)$.
If $\langle P, \gamma, \mathcal{C}\rangle \vdash_{UF} (G_1, u_1) \Leftrightarrow_X (G_2, u_2)$ *then* $\langle P, \gamma, \mathcal{C}\rangle \models (G_1, u_1) \Leftrightarrow_X (G_2, u_2)$.

In the following example we prove the associativity of list concatenation by using the unfold/fold proof method.

Example 4 (*An Unfold/Fold Proof*) Let us consider again the program *Append* and the replacement law (α), expressing the associativity of list concatenation, presented in Example 2. By applying the unfold/fold proof method we will generate a set $\mathcal{C}$ of constraints such that law (α) holds w.r.t. $\langle Append, \gamma, \mathcal{C}\rangle$.

Step 1. We start off by introducing the following two clauses:

$D_1.\ \ new1(L_1, L_2, L_3, L) \leftarrow a(L_1, L_2, M) \wedge a(M, L_3, L) \qquad w_1$
$D_2.\ \ new2(L_1, L_2, L_3, L) \leftarrow a(L_2, L_3, R) \wedge a(L_1, R, L) \qquad w_2$

Step 2. First, let us construct a transformation sequence starting from $Append \cup \{D_1\}$. By two applications of the unfolding rule, from clause D_1 we derive:

$E_1.\ \ new1([\,], L_2, L_3, L) \leftarrow a(L_2, L_3, L) \qquad w_1 + u_1$
$E_2.\ \ new1([H|T], L_2, L_3, [H|R]) \leftarrow a(T, L_2, M) \wedge a(M, L_3, R) \qquad w_1 + 2u_2$

By folding clause E_2 using clause D_1 we derive:

$E_3.\ \ new1([H|T], L_2, L_3, [H|R]) \leftarrow new1(T, L_2, L_3, R) \qquad u_8$

together with the constraint $u_8 \leq 2u_2$.
Now, let us construct a transformation sequence starting from $Append \cup \{D_2\}$. By unfolding clause D_2 w.r.t. $a(L_1, R, L)$ in its body we get:

$F_1.\ \ new2([\,], L_2, L_3, L) \leftarrow a(L_2, L_3, L) \qquad w_2 + u_1$
$F_2.\ \ new2([H|T], L_2, L_3, [H|R]) \leftarrow a(L_2, L_3, M) \wedge a(T, M, R) \qquad w_2 + u_2$

By a symmetric application of the folding rule using clause D_2, from clause F_2 we get:

$$F_3.\ \ new2([H|T], L_2, L_3, [H|R]) \leftarrow new2(T, L_2, L_3, R) \qquad u_9$$

together with the constraint $u_9 = u_2$.

The final programs $Append \cup \{E_1, E_3\}$ and $Append \cup \{F_1, F_3\}$ are syntactically equivalent via the predicate renaming ρ such that $\rho(new1) = new2$. The transformation sequence $Append \cup \{D_2\} \mapsto \cdots \mapsto Append \cup \{F_1, F_3\}$ is symmetric.

Step 3. The correctness constraint system associated with the transformation sequence $Append \cup \{D_1\} \mapsto \cdots \mapsto Append \cup \{E_1, E_3\}$ is the following:

$$\mathcal{C}_1\colon\ \{u_1 \geq 1,\ u_2 \geq 1,\ w_1 + u_1 \geq 1,\ u_8 \geq 1,\ u_8 \leq 2u_2\}$$

The correctness constraint system associated with the transformation sequence $Append \cup \{D_2\} \mapsto \cdots \mapsto Append \cup \{F_1, F_3\}$ is the following:

$$\mathcal{C}_2\colon\ \{u_1 \geq 1,\ u_2 \geq 1,\ w_2 + u_1 \geq 1,\ u_9 \geq 1,\ u_9 = u_2\}$$

Both $\mathcal{C}_1$ and $\mathcal{C}_2$ are satisfiable and thus, we infer:

$$\langle Append, \gamma, \mathcal{C}_{12}\rangle \vdash_{UF} (a(L_1, L_2, M) \wedge a(M, L_3, L), w_1) \Rightarrow_{\{L_1, L_2, L_3, L\}} (a(L_2, L_3, R) \wedge a(L_1, R, L), w_2)$$

where $\mathcal{C}_{12}$ is the set $\{w_1 + u_1 \geq w_2 + u_1,\ u_8 \geq u_9\} \cup \mathcal{C}_1 \cup \mathcal{C}_2$.

Notice that the constraints $w_1 + u_1 \geq w_2 + u_1$ and $u_8 \geq u_9$ are determined by the two pairs of syntactically equivalent clauses (E_1, F_1) and (E_3, F_3), respectively. By eliminating the unknowns u_8 and u_9, which occur in the proof of law (α) only, and by performing some simple simplifications we get, as anticipated, the following set $\mathcal{C}$ of constraints: $\{u_1 \geq 1,\ u_2 \geq 1,\ w_1 \geq w_2,\ w_2 + u_1 \geq 1\}$. □

2.5 Related Work and Conclusions

We have presented a method for proving the correctness of rule-based logic program transformations in an automatic way. Given a transformation sequence, constructed by using the unfold, fold, and goal replacement

transformation rules, we associate some unknown natural numbers, called weights, with the clauses of the programs in the transformation sequence and we also construct a set of linear constraints that these weights must satisfy to guarantee the total correctness of the transformation sequence. Thus, the correctness of the transformation sequence can be proven in an automatic way by checking that the corresponding set of constraints is satisfiable over the natural numbers. However, it can be shown that our method is incomplete and, in general, it can be shown that there exists no algorithmic method for checking whether or not any given unfold/fold transformation sequence is totally correct.

As already mentioned in the introduction to this chapter, our method is related to the many methods given in the literature for proving the correctness of program transformation by showing that suitable conditions on the transformation sequence hold (see, for instance, [13, 31, 33, 66, 79, 80], for the case of definite logic programs). Among these methods, the one presented in [66] is the most general and it makes use of *clause measures* to express complex conditions on the transformation sequence. The main novelty of our method with respect to [66] is that in [66] clause measures are fixed in advance, independently of the specific transformation sequence under consideration, while by the method proposed in this chapter we automatically generate specific clause measures for each transformation sequence to be proved correct.

Thus, in principle, our method is more powerful than the one presented in [66]. For a more accurate comparison between the two methods, we did some practical experiments. We implemented our method in the MAP transformation system, available at:

`http://www.iasi.cnr.it/~proietti/system.html`

and we worked out some transformation examples taken from the literature. Our system runs on SICStus Prolog (v. 3.12.5) and for the satisfiability of the sets of constraints over the natural numbers it uses the *clpq* SICStus library.

By using our system we did the transformation examples presented in this chapter (see Examples 1, 3, and 4) and the following examples taken from the literature: (i) the *Adjacent* program which checks whether or not

two elements have adjacent occurrences in a list [33], (ii) the *Equal Frontiers* program which checks whether or not the frontiers of two binary trees are equal [17, 80], (iii) a program for solving the N-queens problem [71], (iv) the *In_Correct_Position* program taken from [31], and (v) the program that encodes a liveness property of an n-bit shift register [66]. Even in the most complex derivation we carried out, that is, the *Equal Frontiers* example taken from [80], consisting of 97 transformation steps (see Appendix A), the system checked the total correctness of the transformation within milliseconds. For making that derivation we also had to apply several replacement laws which were proved correct by using the unfold/fold proof method described in Section 2.4.

In Appendix A we present a demonstration run of our implementation of the method for the automatic generation of correctness proofs for program transformations presented in this chapter. In particular, we give a demonstration for the examples number (i), (ii), (iv), and (v).

Chapter 3

Proving Properties of Constraint Logic Programs by Eliminating Existential Variables

The program transformation technique, first introduced in [17], can be used as a means of proving program properties. In particular, it has been shown that unfold/fold transformations introduced in [17, 79] can be used to prove several kinds of program properties, such as equivalences of functions defined by recursive equation programs [20, 36], equivalences of predicates defined by logic programs [54], first order properties of predicates defined by stratified logic programs [55], and temporal properties of concurrent systems [26, 67]. In this chapter we consider stratified logic programs *with constraints* and we propose a method based on unfold/fold transformations to prove first order properties of these programs.

The main reason that motivates our method is that transformation techniques may serve as a way of eliminating *existential variables* (that is, variables which occur in the body of a clause and not in its head) and the consequent quantifier elimination can be exploited to prove first order for-

mulas. Quantifier elimination is a well established technique for theorem proving in first order logic [62] and one of its applications is Tarski's decision procedure for the theory of the field of reals. However, no quantifier elimination method has been developed so far to prove formulas within theories defined by constraint logic programs, where the constraints are themselves formulas of the theory of reals. Consider, for instance, the following constraint logic program which defines the membership relation for finite lists of reals:

$$\begin{array}{ll} \mathit{Member}: & \mathit{member}(X, [Y|L]) \leftarrow X = Y \\ & \mathit{member}(X, [Y|L]) \leftarrow \mathit{member}(X, L) \end{array}$$

Suppose we want to show that every finite list of reals has an upper bound:

$$\varphi: \quad \forall L\, \exists U\, \forall X\, (\mathit{member}(X, L) \rightarrow X \leq U)$$

Tarski's quantifier elimination method cannot help in this case, because the membership relation is not defined in the language of the theory of reals. The transformational technique we propose in this chapter, proves the formula φ in two steps. In the first step we transform φ into clause form by applying a variant of the Lloyd-Topor transformation [43], thereby deriving the following clauses:

$$\begin{array}{lll} \mathit{Prop}_1: & 1. & \mathit{prop} \leftarrow \neg p \\ & 2. & p \leftarrow \mathit{list}(L) \wedge \neg q(L) \\ & 3. & q(L) \leftarrow \mathit{list}(L) \wedge \neg r(L, U) \\ & 4. & r(L, U) \leftarrow X > U \wedge \mathit{list}(L) \wedge \mathit{member}(X, L) \end{array}$$

where $\mathit{list}(L)$ holds iff L is a finite list of reals. The predicate *prop* is equivalent to φ in the sense that $M(\mathit{Member}) \models \varphi$ iff $M(\mathit{Member} \cup \mathit{Prop}_1) \models \mathit{prop}$, where $M(P)$ denotes the perfect model of a stratified constraint logic program P. In the second step, we eliminate the existential variables by extending to constraint logic programs the techniques presented in [60] in the case of definite logic programs. For instance, the existential variable X occurring in the body of the above clause 4, is eliminated by applying the unfolding and folding rules and transforming that clause into the following two clauses: $r([X|L], U) \leftarrow X > U \wedge \mathit{list}(L)$ and $r([X|L], U) \leftarrow r(L, U)$. By

iterating the transformation process, we eliminate all existential variables and we derive the following program which defines the predicate *prop*:

$Prop_2$: 1. $prop \leftarrow \neg p$
2′. $p \leftarrow p_1$
3′. $p_1 \leftarrow p_1$

Now, $Prop_2$ is a propositional program and has a *finite* perfect model, which is $\{prop\}$. Since all transformations we have made can be shown to preserve the perfect model, we have that $M(Member) \models \varphi$ iff $M(Prop_2) \models prop$ and, thus, we have completed the proof of φ.

The main contribution of this chapter is the proposal of a proof method for showing that a closed first order formula φ holds in the perfect model of a stratified constraint logic program P, that is, $M(P) \models \varphi$. Our proof method is based on program transformations which eliminate existential variables.

The chapter is organized as follows. In Section 3.1 we consider a class of constraint logic programs, called *lr-programs* (*lr* stands for lists of reals), which is Turing complete and for which our proof method is fully automatic. Those programs manipulate finite lists of reals with constraints which are linear equations and inequations over reals. In Section 3.2 we present the transformation strategy which defines our proof method and we prove its soundness. Due to the undecidability of the first order properties of *lr*-programs, our proof method is necessarily incomplete. Some experimental results obtained by using a prototype implementation are presented in Section 3.4. Finally, in Section 3.5 we discuss related work in the field of program transformation and theorem proving.

3.1 Constraint Logic Programs over Lists of Reals

We assume that the reals are defined by the usual mathematical structure $\mathcal{R} = \langle R, 0, 1, +, \cdot, \leq \rangle$. In order to specify programs and formulas, we use a *typed* first order language [43] with two types: (i) *real*, denoting the set of reals, and (ii) *list of reals* (or *list*, for short), denoting the set of finite lists of reals.

We assume that every element of R is a constant of type *real*. A term p of type *real* is defined as:

$p ::= a \mid X \mid p_1 + p_2 \mid a \cdot X$

where a is a real number and X is a variable of type *real*. We also write aX, instead of $a \cdot X$. A term of type *real* will also be called a *linear polynomial*. An *atomic constraint* is a formula of the form:

$p_1 = p_2$, or $p_1 < p_2$, or $p_1 \leq p_2$

where p_1 and p_2 are linear polynomials. We also write $p_1 > p_2$ and $p_1 \geq p_2$, instead of $p_2 < p_1$ and $p_2 \leq p_1$, respectively. A *constraint* is a finite conjunction of atomic constraints. A *first order formula over reals* is a first order formula constructed out of atomic constraints by using the usual connectives and quantifiers (i.e., $\neg, \wedge, \vee, \rightarrow, \exists, \forall$). By $F_{\mathcal{R}}$ we will denote the set of first order formulas over reals. A term l of type *list* is defined as:

$l ::= L \mid [\,] \mid [p \,|\, l\,]$

where L is a variable of type *list* and p is a linear polynomial. A term of type *list* will also be called a *list*. An *atom* is a formula of the form $r(t_1, \ldots, t_n)$ where r is an n-ary predicate symbol (with $n \geq 0$ and $r \notin \{=, <, \leq\}$) and, for $i = 1, \ldots, n$, t_i is either a linear polynomial or a list. An atom is *linear* if each variable occurs in it at most once. A *literal* is either an atom (i.e., a positive literal) or a negated atom (i.e., a negative literal). A *clause* C is a formula of the form: $A \leftarrow c \wedge L_1 \wedge \ldots \wedge L_m$, where: (i) A is an atom, (ii) c is a constraint, and (iii) $L_1, \ldots, L_m$ are literals. A is called the *head* of the clause, denoted $hd(C)$, and $c \wedge L_1 \wedge \ldots \wedge L_m$ is called the *body* of the clause, denoted $bd(C)$. A *constraint logic program over lists of reals*, or simply a *program*, is a set of clauses. A program is *stratified* if no predicate depends negatively on itself [3]. Given a term or a formula f, $vars(f)$ denotes the set of variables occurring in f. Given a clause C, a variable V is said to be an *existential variable* of C if $V \in vars(bd(C)) - vars(hd(C))$.

The *definition* of a predicate p in a program P, denoted by $Def(p, P)$, is the set of the clauses of P whose head predicate is p. The *extended definition* of p in P, denoted by $Def^*(p, P)$, is the union of the definition of p and the definitions of all predicates in P on which p depends (positively

or negatively). A program is *propositional* if every predicate occurring in the program is 0-ary. Obviously, if P is a propositional program then, for every predicate p, $M(P) \models p$ is decidable.

Definition 4 (*lr*-program) Let X denote a variable of type *real*, L a variable of type *list*, p a linear polynomial, r_1 and r_2 two predicate symbols, and c a constraint. An *lr-clause* is a clause defined as follows:

$$\begin{array}{lllll} \textit{head term}: & h & ::= & X \mid [\,] \mid [X|L] \\ \textit{body term}: & b & ::= & p \mid L \\ \textit{lr-clause}: & C & ::= & r_1(h_1, \ldots, h_k) \leftarrow c \\ & & & \mid\ r_1(h_1, \ldots, h_k) \leftarrow c \wedge r_2(b_1, \ldots, b_m) \\ & & & \mid\ r_1(h_1, \ldots, h_k) \leftarrow c \wedge \neg r_2(b_1, \ldots, b_m) \end{array}$$

where: (i) $vars(p) \neq \emptyset$, (ii) $r_1(h_1, \ldots, h_k)$ is a linear atom, and (iii) clause C has no existential variables. An *lr-program* is a finite set of *lr*-clauses.

We assume that the following *lr*-clauses belong to every *lr*-program (but we will omit them when writing *lr*-programs):

$$\begin{array}{l} \mathit{list}([\,]) \leftarrow \\ \mathit{list}([X|L]) \leftarrow \mathit{list}(L) \end{array}$$

The specific syntactic form of *lr*-programs is required for the automation of the transformation strategy we will introduce in Section 3.2. Here is an *lr*-program:

$$\begin{array}{ll} P_1: & \mathit{sumlist}([\,], Y) \leftarrow Y = 0 \\ & \mathit{sumlist}([X|L], Y) \leftarrow \mathit{sumlist}(L,\ Y - X) \\ & \mathit{haspositive}([X|L]) \leftarrow X > 0 \\ & \mathit{haspositive}([X|L]) \leftarrow \mathit{haspositive}(L) \end{array}$$

The following definition introduces the class of programs and formulas which can be given in input to our proof method.

Definition 5 (Admissible Pair) Let P be an *lr*-program and φ a closed first order formula with no other connectives and quantifiers besides $\neg, \wedge$, and $\exists$. We say that $\langle P, \varphi \rangle$ is an *admissible pair* if: (i) every predicate symbol occurring in φ and different from $\leq, <, =$, also occurs in P, (ii) every

predicate of arity n (>0) occurring in P and different from $\leq, <, =$, has at least one argument of type *list*, and (iii) for every proper subformula σ of φ, if σ is of the form $\neg\psi$, then either σ is a formula in $F_{\mathcal{R}}$ or σ has a free occurrence of a variable of type *list*.

Conditions (ii) and (iii) of Definition 5 are needed to guarantee the soundness of our proof method (see Theorem 9).

Example 5 Let us consider the above program P_1 defining the predicates *sumlist* and *haspositive*, and the formula

$\pi: \quad \forall L\, \forall Y\, ((sumlist(L, Y) \wedge Y > 0) \rightarrow haspositive(L))$

which expresses the fact that if the sum of the elements of a list is positive then the list has at least one positive member. This formula can be rewritten as:

$\pi_1: \quad \neg\exists L\, \exists Y\, (sumlist(L, Y) \wedge Y > 0 \wedge \neg haspositive(L))$

The pair $\langle P_1, \pi_1\rangle$ is admissible. Indeed, the only proper subformula of π_1 of the form $\neg\psi$ is $\neg haspositive(L)$ and the free variable L is of type *list*. □

In order to define the semantics of our logic programs we consider $\mathcal{LR}$-interpretations where: (i) the type *real* is mapped to the set of reals, (ii) the type *list* is mapped to the set of lists of reals, and (iii) the symbols $+$, $\cdot$, $=$, $<$, $\leq$, [ ], and $[\_|\_]$ are mapped to the usual corresponding operations and relations on reals and lists of reals. The semantics of a stratified logic program P is assumed to be its *perfect* $\mathcal{LR}$*-model* $M(P)$, which is defined similarly to the perfect model of a stratified logic program [3, 45, 61] by considering $\mathcal{LR}$-interpretations, instead of Herbrand interpretations. Note that for every formula $\varphi \in F_{\mathcal{R}}$, we have that $\mathcal{R} \models \varphi$ iff for any $\mathcal{LR}$-interpretation $\mathcal{I}$, $\mathcal{I} \models \varphi$.

Now we present a transformation, called *Clause Form Transformation*, that allows us to derive stratified logic programs starting from formulas, called *statements*, of the form: $A \leftarrow \beta$, where A is an atom and β is a typed first order formula. Our transformation is a variant of the transformation proposed by Lloyd and Topor in [43]. When applying the Clause Form Transformation, we will use the following well known property which

guarantees that existential quantification and negation can always be eliminated from first order formulas on reals.

Lemma 6 (Variable Elimination) *For any formula $\varphi \in F_{\mathcal{R}}$ there exist n (≥ 0) constraints $c_1, \ldots, c_n$ such that:* (i) $\mathcal{R} \models \forall(\varphi \leftrightarrow (c_1 \vee \ldots \vee c_n))$, *and* (ii) *every variable in vars$(c_1 \vee \ldots \vee c_n)$ occurs free in φ.*

In what follows we write $C[\gamma]$ to denote a formula where the subformula γ occurs as an *outermost conjunct*, that is, $C[\gamma] = \gamma_1 \wedge \gamma \wedge \gamma_2$ for some (possibly empty) conjunctions γ_1 and γ_2.

Clause Form Transformation.
Input: A statement S whose body has no other connectives and quantifiers besides $\neg, \wedge$, and $\exists$. *Output*: A set of clauses denoted $CFT(S)$.
(*Step* A) Starting from S, repeatedly apply the following rules A.1–A.5 until a set of clauses is generated.
(A.1) If $\gamma \in F_{\mathcal{R}}$ and γ is *not* a constraint, then replace $A \leftarrow C[\gamma]$ by the n statements $A \leftarrow C[c_1], \ldots, A \leftarrow C[c_n]$, where $c_1 \vee \ldots \vee c_n$, with $n \geq 0$, is a disjunction of constraints which is equivalent to γ. (The existence of such a disjunction is guaranteed by Lemma 6 above.)
(A.2) If $\gamma \notin F_{\mathcal{R}}$ then replace $A \leftarrow C[\neg\neg\gamma]$ by $A \leftarrow C[\gamma]$.
(A.3) If $\gamma \wedge \delta \notin F_{\mathcal{R}}$ then replace the statement $A \leftarrow C[\neg(\gamma \wedge \delta)]$ by the two statements $A \leftarrow C[\neg newp(V_1, \ldots, V_k)]$ and $newp(V_1, \ldots, V_k) \leftarrow \gamma \wedge \delta$, where *newp* is a new predicate and $V_1, \ldots, V_k$ are the variables which occur free in $\gamma \wedge \delta$.
(A.4) If $\gamma \notin F_{\mathcal{R}}$ then replace the statement $A \leftarrow C[\neg \exists V\, \gamma]$ by the two statements $A \leftarrow C[\neg newp(V_1, \ldots, V_k)]$ and $newp(V_1, \ldots, V_k) \leftarrow \gamma$, where *newp* is a new predicate and $V_1, \ldots, V_k$ are the variables which occur free in $\exists V\, \gamma$.
(A.5) If $\gamma \notin F_{\mathcal{R}}$ then replace $A \leftarrow C[\exists V\, \gamma]$ by $A \leftarrow C[\gamma\{V/V_1\}]$, where V_1 is a new variable.
(*Step* B) For every clause $A \leftarrow c \wedge G$ such that $L_1, \ldots, L_k$ are the variables of type *list* occurring in G, replace $A \leftarrow c \wedge G$ by $A \leftarrow c \wedge list(L_1) \wedge \ldots \wedge list(L_k) \wedge G$.

Example 6 The set $CFT(prop_1 \leftarrow \pi_1)$, where π_1 is the formula given in Example 5, consists of the following two clauses:

$D_2:\ prop_1 \leftarrow \neg new_1$
$D_1:\ new_1 \leftarrow Y > 0 \wedge list(L) \wedge sumlist(L, Y) \wedge \neg haspositive(L)$

(The subscripts of the names of these clauses follow the bottom-up order in which they will be processed by the UF_{lr} strategy, that we will introduce in the following.) □

By construction, we have that if $\langle P, \varphi\rangle$ is an admissible pair and *prop* is a new predicate symbol, then $P \cup CFT(prop \leftarrow \varphi)$ is a stratified program. The Clause Form Transformation is correct with respect to the perfect $\mathcal{LR}$-model semantics, as stated by the following theorem.

Theorem 7 (Correctness of CFT) *Let $\langle P, \varphi\rangle$ be an admissible pair. Then, $M(P) \models \varphi$ iff $M(P \cup CFT(prop \leftarrow \varphi)) \models prop$.*

In general, a clause in $CFT(prop \leftarrow \varphi)$ is *not* an *lr*-clause because, indeed, existential variables may occur in its body. The clauses of $CFT(prop \leftarrow \varphi)$ are called *typed-definitions*. They are defined as follows.

Definition 6 (Typed-Definition, Hierarchy) A *typed-definition* is a clause of the form: $r(V_1, \ldots, V_n) \leftarrow c \wedge list(L_1) \wedge \ldots \wedge list(L_k) \wedge G$ where: (i) $V_1, \ldots, V_n$ are distinct variables of type *real* or *list*, and (ii) $L_1, \ldots, L_k$ are the variables of type *list* that occur in G. A sequence $\langle D_1, \ldots, D_n\rangle$ of typed-definitions is said to be a *hierarchy* if for $i = 1, \ldots, n$, the predicate of $hd(D_i)$ does not occur in $\{bd(D_1), \ldots, bd(D_i)\}$.

One can show that given a closed formula φ, the set $CFT(prop \leftarrow \varphi)$ of clauses can be ordered as a hierarchy $\langle D_1, \ldots, D_n\rangle$ of typed-definitions such that $Def(prop, \{D_1, \ldots, D_n\}) = \{D_k, D_{k+1}, \ldots, D_n\}$, for some k with $1 \leq k \leq n$.

3.2 The Unfold/Fold Proof Method

In this section we present the transformation strategy, called UF_{lr} (Unfold/Fold strategy for *lr*-programs), which defines our proof method for proving properties of *lr*-programs. Our strategy applies in an automatic way the transformation rules for stratified constraint logic programs presented in [29]. In particular, the UF_{lr} strategy makes use of the definition introduction, (positive and negative) unfolding, (positive) folding, and constraint replacement rules. (These rules extend the ones proposed in [24, 45] where the unfolding of a clause with respect to a negative literal is not permitted.)

Given an admissible pair $\langle P, \varphi \rangle$, let us consider the stratified program $P \cup CFT(prop \leftarrow \varphi)$. The goal of our UF_{lr} strategy is to derive a program *TransfP* such that $Def^*(prop, TransfP)$ is propositional and, thus, $M(TransfP) \models prop$ is decidable. We observe that, in order to achieve this goal, it is enough that the derived program *TransfP* is an *lr*-program, as stated by the following lemma, which follows directly from Definition 4.

Lemma 7 *Let P be an lr-program and p be a predicate occurring in P. If p is 0-ary then $Def^*(p, P)$ is a propositional program.*

As already said, the clauses in $CFT(prop \leftarrow \varphi)$ form a hierarchy $\langle D_1, \ldots, D_n \rangle$ of typed-definitions. The UF_{lr} strategy consists in transforming, for $i = 1, \ldots, n$, clause D_i into a set of *lr*-clauses. The transformation of D_i is performed by applying the following three substrategies, in this order: (i) *unfold*, which unfolds D_i with respect to the positive and negative literals occurring in its body, thereby deriving a set *Cs* of clauses, (ii) *replace-constraints*, which replaces the constraints appearing in the clauses of *Cs* by equivalent ones, thereby deriving a new set *Es* of clauses, and (iii) *define-fold*, which introduces a set *NewDefs* of new typed-definitions (which are not necessarily *lr*-clauses) and folds all clauses in *Es*, thereby deriving a set *Fs* of *lr*-clauses. Then each new definition in *NewDefs* is transformed by applying the above three substrategies, and the whole UF_{lr} strategy terminates when no new definitions are introduced. The sub-

strategies *unfold*, *replace-constraints*, and *define-fold* will be described in detail below.

The UF_{lr} Transformation Strategy.
Input: An *lr*-program P and a hierarchy $\langle D_1, \ldots, D_n \rangle$ of typed-definitions.
Output: A set *Defs* of typed-definitions including $D_1, \ldots, D_n$, and an *lr*-program *TransfP* such that $M(P \cup Defs) = M(TransfP)$.

TransfP := P; *Defs* := $\{D_1, \ldots, D_n\}$;
for $i = 1, \ldots, n$ **do** *InDefs* := $\{D_i\}$;
  while *InDefs* $\neq \emptyset$ **do**
    unfold(*InDefs*, *TransfP*, *Cs*);
    replace-constraints(*Cs*, *Es*);
    define-fold(*Es*, *Defs*, *NewDefs*, *Fs*);
    TransfP := *TransfP* $\cup$ *Fs*;
    Defs := *Defs* $\cup$ *NewDefs*;
    InDefs := *NewDefs*;
  end-while;
  eval-props: for each predicate p such that $Def^*(p, TransfP)$ is propositional,
    if $M(TransfP) \models p$
      then *TransfP* := $(TransfP - Def(p, TransfP)) \cup \{p \leftarrow\}$
      else *TransfP* := $(TransfP - Def(p, TransfP))$
end-for

Our assumption that $\langle D_1, \ldots, D_n \rangle$ is a hierarchy ensures that, when transforming clause D_i, for $i = 1, \ldots, n$, we only need the clauses obtained after the transformation of $D_1, \ldots, D_{i-1}$. These clauses are those of the current value of *TransfP*.

The following *unfold* substrategy transforms a set *InDefs* of typed-definitions by first applying the unfolding rule with respect to each positive literal in the body of a clause and then applying the unfolding rule with

respect to each negative literal in the body of a clause. In the sequel, we will assume that the conjunction operator $\wedge$ is associative, commutative, idempotent, and with neutral element *true*. In particular, the order of the conjuncts will *not* be significant.

The *unfold* Substrategy.
Input: An *lr*-program *Prog* and a set *InDefs* of typed-definitions.
Output: A set *Cs* of clauses.

Initially, no literal in the body of a clause of *InDefs* is marked as 'unfolded'.

Positive Unfolding:

while there exists a clause C in *InDefs* of the form $H \leftarrow c \wedge G_L \wedge A \wedge G_R$, where A is an atom which is not marked as 'unfolded' **do**

Let C_1: $K_1 \leftarrow c_1 \wedge B_1$, ..., C_m: $K_m \leftarrow c_m \wedge B_m$ be all clauses of program *Prog* (where we assume $vars(Prog) \cap vars(C) = \emptyset$) such that, for $i = 1, \ldots, m$, (i) there exists a most general unifier ϑ_i of A and K_i, and (ii) the constraint $(c \wedge c_i)\vartheta_i$ is satisfiable. Let U be the following set of clauses:

$U = \{(H \leftarrow c \wedge c_1 \wedge G_L \wedge B_1 \wedge G_R)\vartheta_1, \ldots, (H \leftarrow c \wedge c_m \wedge G_L \wedge B_m \wedge G_R)\vartheta_m\}$

Let W be the set of clauses derived from U by removing all clauses of the form

$H \leftarrow c \wedge G_L \wedge A \wedge \neg A \wedge G_R$

Inherit the markings of the literals in the body of the clauses of W from those of C, and mark as 'unfolded' the literals $B_1\vartheta_1, \ldots, B_m\vartheta_m$;

$InDefs := (InDefs - \{C\}) \cup W$;

end-while;

Negative Unfolding:

while there exists a clause C in *InDefs* of the form $H \leftarrow c \wedge G_L \wedge \neg A \wedge G_R$, where $\neg A$ is a literal which is not marked as 'unfolded' **do**

Let C_1: $K_1 \leftarrow c_1 \wedge B_1, \ldots, C_m$: $K_m \leftarrow c_m \wedge B_m$ be all clauses of program *Prog* (where we assume that $vars(Prog) \cap vars(C) = \emptyset$) such that, for $i = 1, \ldots, m$, there exists a most general unifier ϑ_i of A and K_i. By our assumptions on *Prog* and on the initial value of *InDefs*, and as a result of

the previous *Positive Unfolding* phase, we have that, for $i=1,\ldots,m$, B_i is either the empty conjunction *true* or a literal and $A=K_i\vartheta_i$. Let U be the following set of statements:

$$U = \{H \leftarrow c \wedge d_1\vartheta_1 \wedge \ldots \wedge d_m\vartheta_m \wedge G_L \wedge L_1\vartheta_1 \wedge \ldots \wedge L_m\vartheta_m \wedge G_R \mid$$
(i) for $i=1,\ldots,m$, either $(d_i=c_i$ and $L_i=\neg B_i)$
or $(d_i=\neg c_i$ and $L_i=true)$, and
(ii) $c \wedge d_1\vartheta_1 \wedge \ldots \wedge d_m\vartheta_m$ is satisfiable$\}$

Let W be the set of clauses derived from U by applying as long as possible the following rules:

- remove $H \leftarrow c \wedge G_L \wedge \neg true \wedge G_R$ and $H \leftarrow c \wedge G_L \wedge A \wedge \neg A \wedge G_R$
- replace $\neg\neg A$ by A, $\neg(p_1 \leq p_2)$ by $p_2 < p_1$, and $\neg(p_1 < p_2)$ by $p_2 \leq p_1$
- replace $H \leftarrow c_1 \wedge \neg(p_1 = p_2) \wedge c_2 \wedge G$
  by $H \leftarrow c_1 \wedge p_1 < p_2 \wedge c_2 \wedge G$ and $H \leftarrow c_1 \wedge p_2 < p_1 \wedge c_2 \wedge G$

Inherit the markings of the literals in the body of the clauses of W from those of C, and mark as 'unfolded' the literals $L_1\vartheta_1, \ldots, L_m\vartheta_m$;

$InDefs := (InDefs - \{C\}) \cup W$;

end-while;

$Cs := InDefs$.

Negative Unfolding is best explained through an example. Let us consider a program consisting of the clauses C: $H \leftarrow c \wedge \neg A$, $A \leftarrow c_1 \wedge B_1$, and $A \leftarrow c_2 \wedge B_2$. The negative unfolding of C w.r.t. $\neg A$ gives us the following four clauses:

$H \leftarrow c \wedge \neg c_1 \wedge \neg c_2$ $\qquad$ $H \leftarrow c \wedge \neg c_1 \wedge c_2 \wedge \neg B_2$
$H \leftarrow c \wedge c_1 \wedge \neg c_2 \wedge \neg B_1$ $\qquad$ $H \leftarrow c \wedge c_1 \wedge c_2 \wedge \neg B_1 \wedge \neg B_2$

whose conjunction is equivalent to $H \leftarrow c \wedge \neg((c_1 \wedge B_1) \vee (c_2 \wedge B_2))$.

Example 7 Let us consider the program-property pair $\langle P_1, \pi_1\rangle$ of Example 5. In order to prove that $M(P_1) \models \pi_1$, we apply the UF_{lr} strategy starting from the hierarchy $\langle D_1, D_2\rangle$ of typed-definitions of Example 6. During the first execution of the body of the for-loop of that strategy, the

unfold substrategy is applied, as we now indicate, by using as input the program P_1 and the set $\{D_1\}$ of clauses.
Positive Unfolding. By unfolding clause D_1 w.r.t. $list(L)$ and then unfolding the resulting clauses w.r.t. $sumlist(L, Y)$, we get:

C_1: $new_1 \leftarrow Y > 0 \wedge list(L) \wedge sumlist(L, Y - X) \wedge \neg haspositive([X|L])$

Negative Unfolding. By unfolding clause C_1 w.r.t. $\neg haspositive([X|L])$, we get:

C_2: $new_1 \leftarrow Y > 0 \wedge X \leq 0 \wedge list(L) \wedge sumlist(L, Y - X) \wedge \neg haspositive(L)$

The correctness of the *unfold* substrategy follows from the fact that the positive and negative unfoldings are performed according to the rules presented in [29]. The termination of that substrategy is due to the fact that the number of literals which are not marked as 'unfolded' and which occur in the body of a clause, decreases when that clause is unfolded. Thus, we have the following result.

Lemma 8 *Let Prog be an lr-program and let InDefs be a set of typed-definitions such that the head predicates of the clauses of InDefs do not occur in Prog. Then, given the inputs Prog and InDefs, the unfold substrategy terminates and returns a set Cs of clauses such that* $M(Prog \cup InDefs) = M(Prog \cup Cs)$.

The *replace-constraints* substrategy derives from a set *Cs* of clauses a new set *Es* of clauses by applying equivalences between existentially quantified disjunctions of constraints. We use the following two rules: *project* and *clause split*.

Given a clause $H \leftarrow c \wedge G$, the *project* rule eliminates all variables that occur in c and do not occur elsewhere in the clause. Thus, *project* returns a new clause $H \leftarrow d \wedge G$ such that $\mathcal{R} \models \forall((\exists X_1 \ldots \exists X_k\, c) \leftrightarrow d)$, where: (i) $\{X_1, \ldots, X_k\} = vars(c) - vars(\{H, G\})$, and (ii) $vars(d) \subseteq vars(c) - \{X_1, \ldots, X_k\}$. In our prototype theorem prover (see Section 3.4), the *project* rule is implemented by using a variant of the Fourier-Motzkin Elimination algorithm [2].

The *clause split* rule replaces a clause C by two clauses C_1 and C_2 such that, for $i = 1, 2$, the number of occurrences of existential variables in C_i is less than the number of occurrences of existential variables in C. The clause split rule applies the following property, which expresses the fact that $\langle \mathcal{R}, \leq \rangle$ is a linear order: $\mathcal{R} \models \forall X \, \forall Y \, (X < Y \vee Y \leq X)$. For instance, a clause of the form $H \leftarrow Z \leq X \wedge Z \leq Y \wedge G$, where Z is an existential variable occurring in the conjunction G of literals and X and Y are not existential variables, is replaced by the two clauses $H \leftarrow Z \leq X \wedge X < Y \wedge G$ and $H \leftarrow Z \leq Y \wedge Y \leq X \wedge G$. The decrease of the number of occurrences of existential variables guarantees that we can apply the clause split rule a finite number of times only.

The *replace-constraints* Substrategy.
Input: A set *Cs* of clauses. *Output*: A set *Es* of clauses.

• *Introduce Equations.* (A) From *Cs* we derive a new set R_1 of clauses by applying as long as possible the following two rules, where p denotes a linear polynomial which is not a variable, and Z denotes a fresh new variable:

(R.1) $H \leftarrow c \wedge G_L \wedge r(\ldots, p, \ldots) \wedge G_R$ is replaced by
$H \leftarrow c \wedge Z = p \wedge G_L \wedge r(\ldots, Z, \ldots) \wedge G_R$

(R.2) $H \leftarrow c \wedge G_L \wedge \neg r(\ldots, p, \ldots) \wedge G_R$ is replaced by
$H \leftarrow c \wedge Z = p \wedge G_L \wedge \neg r(\ldots, Z, \ldots) \wedge G_R$

(B) From R_1 we derive a new set R_2 of clauses by applying to every clause C in R_1 the following rule. Let C be of the form $H \leftarrow c \wedge G$. Suppose that $\mathcal{R} \models \forall (c \leftrightarrow (X_1 = p_1 \wedge X_n = p_n \wedge d))$, where: (i) $X_1, \ldots, X_n$ are existential variables of C, (ii) $vars(X_1 = p_1 \wedge \ldots \wedge X_n = p_n \wedge d) \subseteq vars(c)$, (iii) $\{X_1, \ldots, X_n\} \cap vars(\{p_1, \ldots, p_n, d\}) = \emptyset$. Then we replace C by $H \leftarrow X_1 = p_1 \wedge \ldots \wedge X_n = p_n \wedge d \wedge G$.

• *Project.* We derive a new set R_3 of clauses by applying to every clause in R_2 the *project* rule.

• *Clause Split.* From R_3 we derive a new set R_4 of clauses by applying as long as possible the following rule. Let C be a clause of the form $H \leftarrow$

$c_1 \wedge c_2 \wedge c \wedge G$ (modulo commutativity of $\wedge$). Let E be the set of existential variables of C. Let $X \in E$ and let d_1 and d_2 be two inequations such that $\mathcal{R} \models \forall((c_1 \wedge c_2) \leftrightarrow (d_1 \wedge d_2))$. Suppose that: (i) $d_1 \wedge d_2$ is of one of the following six forms:

$$\begin{array}{lll} X \leq p_1 \wedge X \leq p_2 & X \leq p_1 \wedge X < p_2 & X < p_1 \wedge X < p_2 \\ p_1 \leq X \wedge p_2 \leq X & p_1 \leq X \wedge p_2 < X & p_1 < X \wedge p_2 < X \end{array}$$

and (ii) $(vars(p_1) \cup vars(p_2)) \cap E = \emptyset$.
Then C is replaced by the following two clauses: C_1: $H \leftarrow d_1 \wedge p_1 < p_2 \wedge c \wedge G$ and C_2: $H \leftarrow d_2 \wedge p_2 \leq p_1 \wedge c \wedge G$, and then each clause in $\{C_1, C_2\}$ with an unsatisfiable constraint in its body is removed.

• *Eliminate Equations.* From R_4 we derive the new set Es of clauses by applying to every clause C in R_4 the following rule. If C is of the form $H \leftarrow X_1 = p_1 \wedge \ldots \wedge X_n = p_n \wedge d \wedge G$ where $\{X_1, \ldots, X_n\} \cap vars(\{p_1, \ldots p_n, d\}) = \emptyset$, then C is replaced by $(H \leftarrow d \wedge G)\{X_1/p_1, \ldots, X_n/p_n\}$.

The transformation described at Point (A) of *Introduce Equations* allows us to treat all polynomials occurring in the body of a clause in a uniform way as arguments of constraints. The transformation described at Point (B) of *Introduce Equations* identifies those existential variables which can be eliminated during the final *Eliminate Equations* transformation. That elimination is performed by substituting, for $i = 1, \ldots, n$, the variable X_i by the polynomial p_i.

Example 8 By applying the *replace-constraints* substrategy, clause C_2 of Example 7 is transformed as follows. By introducing equations we get:

C_3: $new_1 \leftarrow Y > 0 \wedge X \leq 0 \wedge Z = Y - X \wedge list(L) \wedge$
$\quad sumlist(L, Z) \wedge \neg haspositive(L)$

Then, by applying the *project* transformation, we get:

C_4: $new_1 \leftarrow Z > 0 \wedge list(L) \wedge sumlist(L, Z) \wedge \neg haspositive(L)$ □

The correctness of the *replace-constraints* substrategy is a straightforward consequence of the fact that the *Introduce Equations*, *Project*, *Clause*

Split, and *Eliminate Equations* transformations are performed by using the rule of *replacement based on laws* presented in [29]. The termination of *Introduce Equations* and *Eliminate Equations* is obvious. The termination of *Project* is based on the termination of the specific algorithm used for variable elimination (e.g., Fourier-Motzkin algorithm). As already mentioned, the termination of *Clause Split* is due to the fact that at each application of this transformation the number of occurrences of existential variables decreases. Thus, we get the following lemma.

Lemma 9 *For any program Prog and set* $Cs \subseteq Prog$ *of clauses, the replace-constraints substrategy with input Cs terminates and returns a set Es of clauses such that* $M(Prog) = M((Prog - Cs) \cup Es)$.

The *define-fold* substrategy eliminates all existential variables in the clauses of the set *Es* obtained after the *unfold* and *replace-constraints* substrategies. This elimination is done by folding all clauses in *Es* that contain existential variables. In order to make these folding steps we use the typed-definitions in *Defs* and, if necessary, we introduce new typed-definitions which we add to the set *NewDefs*.

The *define-fold* Substrategy.
Input: A set *Es* of clauses and a set *Defs* of typed-definitions.
Output: A set *NewDefs* of typed-definitions and a set *Fs* of *lr*-clauses.

Initially, both *NewDefs* and *Fs* are empty.

for each clause C: $H \leftarrow c \wedge G$ in *Es* **do**

if C is an *lr*-clause **then** $Fs := Fs \cup \{C\}$ **else**

• *Define.* Let E be the set of existential variables of C. We consider a clause *NewD* of the form $newp(V_1, \ldots, V_m) \leftarrow d \wedge B$ constructed as follows: (1) let c be of the form $c_1 \wedge c_2$, where $vars(c_1) \cap E = \emptyset$ and for every atomic constraint a occurring in c_2, $vars(a) \cap E \neq \emptyset$; let $d \wedge B$ be the most general (modulo variants) conjunction of constraints and literals such that there exists a substitution ϑ with the following properties: (i) $(d \wedge B)\vartheta = c_2 \wedge G$,

and (ii) for each binding V/p in ϑ, V is a variable not occurring in C, $vars(p) \neq \emptyset$, and $vars(p) \cap E = \emptyset$;

(2) *newp* is a new predicate symbol;

(3) $\{V_1, \ldots, V_m\} = vars(d \wedge B) - E$.

NewD is added to *NewDefs*, unless in *Defs* there exists a typed-definition D which is equal to *NewD*, modulo the name of the head predicate, the names of variables, equivalence of constraints, and the order and multiplicity of literals in the body. If such a clause D belongs to *Defs* and no other clause in *Defs* has the same head predicate as D, then we assume that $NewD = D$.

- *Fold.* Clause C is folded using clause *NewD* as follows:

$Fs := Fs \cup \{H \leftarrow c_1 \wedge newp(V_1, \ldots, V_m)\vartheta\}$.

end-for

Example 9 Let us consider the clause C_4 derived at the end of Example 8. The *Define* phase produces a typed-definition which is a variant of the typed-definition D_1 introduced at the beginning of the application of the strategy (see Example 6). Thus, C_4 is folded using clause D_1, and we get the clause:

C_5: $new_1 \leftarrow new_1$

Let us now describe how the proof of $M(P_1) \models \pi_1$ proceeds. The program *TransfP* derived so far consists of clause C_5 together with the clauses defining the predicates *list*, *sumlist*, and *haspositive*. Thus, $Def^*(new_1, TransfP)$ consists of clause C_5 only, which is propositional and, by *eval-props*, we remove C_5 from *TransfP* because $M(TransfP) \not\models new_1$. The strategy continues by considering the typed definition D_2 (see Example 6). By unfolding D_2 with respect to $\neg new_1$ we get the final program *TransfP*, which consists of the clause $prop_1 \leftarrow$ together with the clauses for *list*, *sumlist*, and *haspositive*. Thus, $M(TransfP) \models prop_1$ and, therefore, $M(P_1) \models \pi_1$. □

The proof of correctness for the *define-fold* substrategy is more complex than the proofs for the other substrategies. The correctness results for the

unfold/fold transformations presented in [29] guarantee the correctness of a folding transformation if each typed-definition used for folding is unfolded w.r.t. a positive literal during the application of the UF_{lr} transformation strategy. The fulfillment of this condition is ensured by the following two facts: (1) by the definition of an admissible pair and by the definition of the Clause Form Transformation, each typed-definition has at least one positive literal in its body (indeed, by Condition (iii) of Definition 5 each negative literal in the body of a typed-definition has at least one variable of type *list* and, therefore, the body of the typed-definition has at least one *list* atom), and (2) in the *Positive Unfolding* phase of the *unfold* substrategy, each typed-definition is unfolded w.r.t. all positive literals.

Note that the set *Fs* of clauses derived by the *define-fold* substrategy is a set of *lr*-clauses. Indeed, by the *unfold* and *replace-constraints* substrategies, we derive a set *Es* of clauses of the form $r(h_1, \ldots, h_k) \leftarrow c \wedge G$, where $h_1, \ldots, h_k$ are head terms (see Definition 4). By folding we derive clauses of the form $r(h_1, \ldots, h_k) \leftarrow c_1 \wedge newp(V_1, \ldots, V_m)\vartheta$, where $vars(c_1 \wedge newp(V_1, \ldots, V_m)\vartheta) \subseteq vars(r(h_1, \ldots, h_k))$, and for $i = 1, \ldots, m$, $vars(V_i\vartheta) \neq \emptyset$ (by the conditions at Points (1)–(3) of the *Define* phase). Hence, all clauses in *Fs* are *lr*-clauses.

The termination of the *define-fold* substrategy is obvious, as each clause is folded at most once. Thus, we have the following result.

Lemma 10 *During the UF_{lr} strategy, if the define-fold substrategy takes as inputs the set Es of clauses and the set Defs of typed-definitions, then this substrategy terminates and returns a set NewDefs of typed-definitions and a set Fs of lr-clauses such that $M(TransfP \cup Es \cup NewDefs) = M(TransfP \cup Fs \cup NewDefs)$.*

By using Lemmata 8, 9, and 10 we get the following correctness result for the UF_{lr} strategy.

Theorem 8 *Let P be an lr-program and $\langle D_1, \ldots, D_n \rangle$ a hierarchy of typed-definitions. Suppose that the UF_{lr} strategy with inputs P and $\langle D_1, \ldots, D_n \rangle$ terminates and returns a set Defs of typed-definitions and a program*

TransfP. *Then*: (i) *TransfP is an lr-program and* (ii) $M(P \cup Defs) = M(TransfP)$.

Now, we are able to prove the soundness of the unfold/fold proof method.

Theorem 9 (Soundness of the Unfold/Fold Proof Method) *Let* $\langle P, \varphi \rangle$ *be an admissible pair and let* $\langle D_1, \ldots, D_n \rangle$ *be the hierarchy of typed-definitions obtained from* $prop \leftarrow \varphi$ *by the Clause Form Transformation. If the* UF_{lr} *strategy with inputs* P *and* $\langle D_1, \ldots, D_n \rangle$ *terminates and returns a program TransfP, then*:

$M(P) \models \varphi \quad iff \quad (prop \leftarrow) \in TransfP$

Proof: By Theorem 7 and Point (ii) of Theorem 8, we have that $M(P) \models \varphi$ iff $M(TransfP) \models prop$. By Point (i) of Theorem 8 and Lemma 7 we have that $Def^*(prop, TransfP)$ is propositional. Since the last step of the UF_{lr} strategy is an application of the *eval-props* transformation, we have that $Def^*(prop, TransfP)$ is either the singleton $\{prop \leftarrow\}$, if $M(TransfP) \models prop$, or the empty set, if $M(TransfP) \not\models prop$. □

3.3 A Complete Example

As an example of application of our transformation strategy for proving properties of constraint logic programs we consider the *lr*-program *Member* and the property φ given in the Introduction. The formula φ is rewritten as follows:

$\varphi_1: \quad \neg\exists L\, \neg\exists U\, \neg\exists X\, (X > U \wedge member(X, L))$

The pair $\langle Member, \varphi_1 \rangle$ is admissible. By applying the Clause Form Transformation starting from the statement $prop \leftarrow \varphi_1$, we get the following clauses:

$D_4: \quad prop \leftarrow \neg p$
$D_3: \quad p \leftarrow list(L) \wedge \neg q(L)$
$D_2: \quad q(L) \leftarrow list(L) \wedge \neg r(L, U)$
$D_1: \quad r(L, U) \leftarrow X > U \wedge list(L) \wedge member(X, L)$

where $\langle D_1, D_2, D_3, D_4 \rangle$ is a hierarchy of typed-definitions. Note that the three nested negations in φ_1 generate the three atoms p, $q(L)$, and $r(L,U)$ with their typed-definitions D_3, D_2, and D_1, respectively. The arguments of p, q, and r are the free variables of the corresponding subformulas of φ_1. For instance, $r(L,U)$ corresponds to the subformula $\exists X\,(X > U \wedge \mathit{member}(X,L))$ which has L and U as free variables. Now we apply the UF_{lr} strategy starting from the program *Member* and the hierarchy $\langle D_1, D_2, D_3, D_4 \rangle$.

- *Execution of the for-loop with* $i = 1$. We have: $\mathit{InDefs} = \{D_1\}$. By unfolding clause D_1 w.r.t. the atoms $\mathit{list}(L)$ and $\mathit{member}(X,L)$ we get:

  1.1 $\quad r([X|T],U) \leftarrow X > U \wedge \mathit{list}(T)$
  1.2 $\quad r([X|T],U) \leftarrow Y > U \wedge \mathit{list}(T) \wedge \mathit{member}(Y,T)$

No replacement of constraints is performed. Then, by folding clause 1.2 using D_1, we get:

  1.3 $\quad r([X|T],U) \leftarrow r(T,U)$

After the *define-fold* substrategy the set *Fs* of clauses is $\{1.1, 1.3\}$, and at this point the program *TransfP* is $\mathit{Member} \cup \{1.1, 1.3\}$. No new definitions are introduced and, thus, $\mathit{InDefs} = \emptyset$ and the while-loop terminates. *eval-props* is not performed because the predicate r is not propositional.

- *Execution of the for-loop with* $i = 2$. We have: $\mathit{InDefs} = \{D_2\}$. We unfold clause D_2 w.r.t. $\mathit{list}(L)$ and $\neg r(L,U)$. Then we introduce the new definition:

  2.1 $\quad q_1(X,T) \leftarrow X \leq U \wedge \mathit{list}(T) \wedge \neg r(T,U)$

and we fold using clause 2.1 (no constraint replacement is performed). We get:

  2.2 $\quad q([\,]) \leftarrow$
  2.3 $\quad q([X|T]) \leftarrow q_1(X,T)$

Since $\mathit{NewDefs} = \mathit{InDefs} = \{2.1\}$ we execute again the body of the while-loop. By unfolding clause 2.1 w.r.t. $\mathit{list}(T)$ and $\neg r(T,U)$, we get:

  2.4 $\quad q_1(X,[\,]) \leftarrow$
  2.5 $\quad q_1(X,[Y|T]) \leftarrow X \leq U \wedge Y \leq U \wedge \mathit{list}(T) \wedge \neg r(T,U)$

By applying *replace-constraints*, clause 2.5 generates the following two clauses:

2.5.1 $q_1(X, [Y|T]) \leftarrow X > Y \wedge X \leq U \wedge \mathit{list}(T) \wedge \neg r(T, U)$
2.5.2 $q_1(X, [Y|T]) \leftarrow X \leq Y \wedge Y \leq U \wedge \mathit{list}(T) \wedge \neg r(T, U)$

By folding clauses 2.5.1 and 2.5.2 using clause 2.1, we get:

2.6 $q_1(X, [Y|T]) \leftarrow X > Y \wedge q_1(X, T)$
2.7 $q_1(X, [Y|T]) \leftarrow X \leq Y \wedge q_1(Y, T)$

At this point the program *TransfP* is

$\mathit{Member} \cup \{1.1, 1.3, 2.2, 2.3, 2.4, 2.6, 2.7\}$

No new definitions are introduced and, thus, the while-loop terminates. *eval-props* is not performed because the predicates q and q_1 are not propositional.

• *Execution of the for-loop with* $i = 3$. We have: $\mathit{InDefs} = \{D_3\}$. By unfolding clause D_3 w.r.t. $\mathit{list}(L)$ and $\neg q(L)$, we get:

3.1 $p \leftarrow \mathit{list}(T) \wedge \neg q_1(X, T)$

No replacement of constraints is performed. The following new definition:

3.2 $p_1 \leftarrow \mathit{list}(T) \wedge \neg q_1(X, T)$

is introduced. Then by folding clause 3.1 using clause 3.2, we get:

3.3 $p \leftarrow p_1$

Since $\mathit{NewDefs} = \mathit{InDefs} = \{3.2\}$ we execute again the body of the while-loop. By unfolding clause 3.2 w.r.t. $\mathit{list}(T)$ and $\neg q_1(X, T)$, we get:

3.4 $p_1 \leftarrow X > Y \wedge \mathit{list}(T) \wedge \neg q_1(X, T)$
3.5 $p_1 \leftarrow X \leq Y \wedge \mathit{list}(T) \wedge \neg q_1(Y, T)$

Since the variable Y occurring in the constraints $X > Y$ and $X \leq Y$ is existential, we apply the *project* rule to clauses 3.4 and 3.5 and we get the following clause:

3.6 $p_1 \leftarrow \mathit{list}(T) \wedge \neg q_1(X, T)$

This clause can be folded using clause 3.2, thereby deriving the following clause:

3.7 $p_1 \leftarrow p_1$

Clauses 3.3 and 3.7 are added to *TransfP*. Since the predicates p and p_1 are both propositional, we execute *eval-props*. We have that:

(i) $M(\mathit{TransfP}) \not\models p_1$ and (ii) $M(\mathit{TransfP}) \not\models p$

Thus, clauses 3.3 and 3.7 are removed from *TransfP*.
Hence,

$\mathit{TransfP} = \mathit{Member} \cup \{1.1, 1.3, 2.2, 2.3, 2.4, 2.6, 2.7\}$.

- *Execution of the for-loop with* $i = 4$. We have: $\mathit{InDefs} = \{D_4\}$. By unfolding clause D_4 w.r.t. $\neg p$, we get the clause: $\mathit{prop} \leftarrow$
This clause shows that, as expected, property φ holds for any finite list of reals.

3.4 Experimental Results

We have implemented our proof method by using the MAP transformation system [46] running under SICStus Prolog on a 900MHz Power PC. Constraint satisfaction and entailment were performed using the *clpr* module of SICStus. Our prototype has automatically proved the properties listed in the following table, where the predicates *member*, *sumlist*, and *haspositive* are defined as shown in the introduction to this chapter and in Section 3.1, and the other predicates have the following meanings: (i) $\mathit{ord}(L)$ holds iff L is a list of the form $[a_1, \ldots, a_n]$ and for $i = 1, \ldots, n-1$, $a_i \leq a_{i+1}$, (ii) $\mathit{sumzip}(L, M, N)$ holds iff L, M, and N are lists of the form $[a_1, \ldots, a_n]$, $[b_1, \ldots, b_n]$, and $[a_1{+}b_1, \ldots, a_n{+}b_n]$, respectively, and (iii) $\mathit{leqlist}(L, M)$ holds iff L and M are lists of the form $[a_1, \ldots, a_n]$ and $[b_1, \ldots, b_n]$, respectively, and for $i = 1, \ldots, n$, $a_i \leq b_i$. We do not write here the *lr*-programs which define the predicates $\mathit{ord}(L)$, $\mathit{sumzip}(L, M, N)$, and $\mathit{leqlist}(L, M)$.

Property	Time
$\forall L\, \exists M\, \forall Y\, (\mathit{member}(Y, L) \rightarrow Y \leq M)$	140 ms
$\forall L\, \forall Y\, ((\mathit{sumlist}(L, Y) \wedge Y > 0) \rightarrow \mathit{haspositive}(L))$	170 ms
$\forall L\, \forall Y\, ((\mathit{sumlist}(L, Y) \wedge Y > 0) \rightarrow \exists X(\mathit{member}(X, L) \wedge X > 0))$	160 ms
$\forall L\, \forall M\, \forall N\, ((\mathit{ord}(L) \wedge \mathit{ord}(M) \wedge \mathit{sumzip}(L, M, N)) \rightarrow \mathit{ord}(N))$	160 ms
$\forall L\, \forall M\, ((\mathit{leqlist}(L, M) \wedge \mathit{sumlist}(L, X) \wedge \mathit{sumlist}(M, Y)) \rightarrow X \leq Y)$	50 ms

3.5 Related Work and Conclusions

We have presented a method for proving first order properties of constraint logic programs based on unfold/fold program transformations, and we have shown that the ability of unfold/fold transformations to eliminate existential variables [60] can be turned into a useful theorem proving method. We have provided a fully automatic strategy for the class of *lr*-programs, which are programs acting on reals and finite lists of reals, with constraints as linear equations and inequations over reals. The choice of lists is actually a simplifying assumption we have made and we believe that the extension of our method to any finitely generated data structure is quite straightforward. However, the use of constraints over the reals is an essential feature of our method, because quantifier elimination from constraints is a crucial subprocedure of our transformation strategy.

The first order properties of *lr*-programs are undecidable (and not even semi-decidable), because one can encode every partial recursive function as an *lr*-program without list arguments. As a consequence our proof method is necessarily incomplete. We have implemented the proof method based of program transformation and we have proved some simple, yet non-trivial, properties. As the experiments show, the performance of our method is encouraging.

Our method is an extension of the method presented in [55] which considers logic programs without constraints. The addition of constraints is a very relevant feature, because it provides more expressive power and, as already mentioned, we may use special purpose theorem provers for checking constraint satisfaction and for quantifier elimination. Our method can also be viewed as an extension of other techniques based on unfold/fold transformations for proving equivalences of predicates [54, 67], and indeed, our method can deal with a class of first order formulas which properly includes equivalences.

Some papers have proposed transformational techniques to prove propositional temporal properties of finite and/or infinite state systems (see, for instance, [26, 41, 67]). Since propositional temporal logic can be encoded in first order logic, in principle these techniques can be viewed as instances of

the unfold/fold proof method presented here. However, it should be noted that the techniques described in [26, 41, 67] have their own peculiarities because they are tailored to the specific problem of verifying concurrent systems.

Finally, we think that a direct comparison of the power of our proof method with that of traditional theorem provers is somewhat inappropriate. The techniques used in those provers are very effective and are the result of a well established line of research (see, for instance, [14] for a survey on the automation of mathematical induction). However, our approach has its novelty and is based on principles which have not been explored in the field of theorem proving. In particular, the idea of making inductive proofs by unfold/fold transformations for eliminating quantifiers, has not yet been investigated within the theorem proving community.

Chapter 4

Folding Transformation Rules for Constraint Logic Programs

Rule-based program transformation is a program development technique which has been first proposed by Burstall and Darlington in the context of functional programming [17], and then it has been extended to logic programming [79] and to other programming paradigms as well (see [53] for an overview).

In this chapter we consider constraint logic programs [32] and the transformation rules presented in [9, 24, 29, 45]. In particular, we focus our investigation on the folding rule, which can be defined (in a declarative style) as follows.

Let (i) H and K be atoms, (ii) c and d be constraints, and (iii) G and B be goals (that is, conjunctions of atoms). Given a clause $\gamma\colon\ H \leftarrow c \wedge G$ and a clause $\delta\colon\ K \leftarrow d \wedge B$, if there exist a constraint e, a substitution ϑ, and a goal R such that $H \leftarrow c \wedge G$ is equivalent to $H \leftarrow e \wedge (d \wedge B)\vartheta \wedge R$ (w.r.t. a given theory of constraints), then γ is transformed into $H \leftarrow e \wedge K\vartheta \wedge R$.

In the literature, the folding rule is presented with respect to a generic theory of constraints and no algorithm is provided to determine whether

the suitable e, ϑ, and R needed for applying the rule exist or not. In this chapter we assume that the constraints are linear equations and inequations over the rational numbers (but the techniques we will present are valid also for the domain of the real numbers, without relevant changes), and we propose an algorithm based on standard linear algebra and term rewriting techniques for computing e, ϑ, and R, if they exist. For instance, let us consider the clauses:

γ: $p(X_1, X_2) \leftarrow Z > 0 \wedge X_2 > Z+1 \wedge X_1 \geq 2X_2 \wedge s(X_1, Z)$

δ: $q(Y_1, Y_2, Y_3) \leftarrow Y_1+Y_3 \geq Y_2 \wedge Y_2 > 0 \wedge Y_2-2 > 2Y_3 \wedge s(Y_1 + Y_3, Y_3)$

and suppose that we want to fold γ using δ. Our folding algorithm computes: (i) the constraint e: $X_2>1 \wedge X_1 \geq 2X_2 \wedge X_1 \geq Y_1 \wedge Y_3+Y_1 = X_1$, (ii) the empty substitution ϑ: $\{\}$, and (iii) the goal R which is the atom *true*, that is, the empty conjunction of atoms. The clause derived by folding γ using δ is:

η: $p(X_1, X_2) \leftarrow X_2>1 \wedge X_1 \geq 2X_2 \wedge X_1 \geq Y_1 \wedge Y_3+Y_1 = X_1 \wedge q(Y_1, Y_2, Y_3)$

In general, there may be zero, one, or more than one triple $\langle e, \vartheta, R\rangle$ that satisfies the conditions for the applicability of the folding rule. For this reason, our folding algorithm is nondeterministic and, for a given pair γ and δ, may compute more than one folded clause η.

Now we want to motivate the introduction of some variants of the folding rule which are useful during program transformation.

In some cases the folding rule is not immediately applicable, but it becomes applicable after applying the *clause splitting* rule [29]. Clause splitting consists in replacing a clause $H \leftarrow c \wedge G$ by two clauses $H \leftarrow c \wedge d_1 \wedge G$ and $H \leftarrow c \wedge d_2 \wedge G$, such that in the given theory of constraints the universal closure of $d_1 \vee d_2$ is equivalent to *true*. In this chapter we propose an extension of the folding rule which is equivalent to several applications of the clause splitting rule, by which from clause γ we derive n clauses $\gamma_1, \ldots, \gamma_n$, followed by n applications of the usual folding rule, by which we fold $\gamma_1, \ldots, \gamma_n$ using clause δ and we derive n clauses $\eta_1, \ldots, \eta_n$. Determining the suitable applications of clause splitting which allow for subsequent applications of the folding rule is a non-trivial task. We modify the folding algorithm provided for the standard folding rule so as to apply in an

automatic way the extended folding rule, which combines clause splitting and folding. Let us consider the following clauses γ and δ to illustrate an application of this extended rule:

γ: $p(X) \leftarrow 2 \geq X \wedge X \geq 1 \wedge Z \geq 1 \wedge 2 \geq Z \wedge q(Z)$
δ: $s(Y) \leftarrow W + \frac{3}{4} \geq Y \wedge Y \geq W - \frac{3}{4} \wedge 4 - W \geq Y \wedge Y \geq 2 - W \wedge q(W)$

It is not possible to fold clause γ using δ, because no constrained subgoal of clause γ is an instance of the body of clause δ. In order to allow folding, our algorithm calculates the following splitting of clause γ:

γ_1: $p(X) \leftarrow 2 \geq X \wedge X \geq 1 \wedge Z \geq 1 \wedge Z \geq \frac{5}{4} \wedge q(Z)$
γ_2: $p(X) \leftarrow 2 \geq X \wedge X \geq 1 \wedge Z \geq 1 \wedge \frac{7}{4} \geq Z \wedge q(Z)$

Now it is possible to fold both clause γ_1 and clause γ_2 using δ. Our algorithm computes: (i) the constraints e_1: $2 \geq X \wedge X \geq 1 \wedge 1 \geq Y$ and e_2: $2 \geq X \wedge X \geq 1 \wedge Y \geq 2$, (ii) the substitutions ϑ_1: $\{W/Z\}$ and ϑ_2: $\{W/Z\}$, and (iii) the goals R_1 and R_2 which are the atom *true*, that is, the empty conjunction of atoms. The clauses derived by folding γ_1 and γ_2 using δ are:

η_1: $p(X) \leftarrow 2 \geq X \wedge X \geq 1 \wedge 1 \geq Y \wedge s(Y)$
η_2: $p(X) \leftarrow 2 \geq X \wedge X \geq 1 \wedge Y \geq 2 \wedge s(Y)$

In some applications of program transformation, the folding rule is applied with the aim of eliminating *existential variables*, that is, variables occurring in the body of a clause but not in the head [57, 60]. We will consider a variant of the folding rule whose application guarantees that the clause derived by folding has no existential variables. Below we will propose an algorithm to deal with this variant of the folding rule. This algorithm exploits a property ensuring that, when we require that an application of the folding rule eliminates the existential variables, the search for a suitable triple $\langle e, \vartheta, R \rangle$ can be performed in a much more restricted set. The following example illustrates an application of this variant of the folding rule:

γ: $p(X_1, X_2) \leftarrow Z > 0 \ \wedge \ X_1 \geq Z + 1 \ \wedge \ X_2 > X_1 \ \wedge \ s(Z)$
δ: $q(Y_1, Y_2) \leftarrow W > 0 \wedge Y_1 - 3 \geq 2W \wedge 2Y_2 > Y_1 + 1 \wedge s(W)$

our folding algorithm computes: (i) the constraint e: $X_1 \geq 1 \wedge X_2 > X_1$, (ii) the substitution ϑ: $\{Y_1/2X_1 + 1, \ Y_2/X_2 + 1, \ W/Z\}$, and (iii) the goal

R which is the atom *true*, that is, the empty conjunction of atoms. The clause derived by folding γ using δ is:

η: $p(X_1, X_2) \leftarrow X_1 \geq 1 \wedge X_2 > X_1 \wedge q(2X_1+1, X_2+1)$

Note that clause η has no existential variables.

The chapter is organized as follows. In Section 4.1 we recall some basic definitions concerning constraint logic programs. In Sections 4.2, 4.3, and 4.4, we introduce the usual folding rule [9, 24, 29, 45] and two variants of this rule. We present algorithms for applying the folding rule and its two variants. We also prove some soundness and completeness results of these algorithms with respect to the declarative specifications of the three folding rules. Finally, in Section 4.5, we discuss related work and suggest some directions for further development.

4.1 Preliminary Definitions

In this section we recall some basic definitions concerning constraint logic programs, where the constraints are conjunctions of linear equations and inequations over rational numbers. As already said, the results presented in the following sections are valid also when the constraints are conjunctions of linear equations and inequations over real numbers. For notions not defined here we refer to [32, 43].

Let us consider a first order language $\mathcal{L}$ as defined in [43]. The language $\mathcal{L}$ is divided into the two languages $\mathcal{L}_u$ and $\mathcal{L}_c$, respectively, of the user-defined formulas and of the constraints. The infinite set *Vars* of variables of $\mathcal{L}$ is partitioned into the infinite sets $Vars_u$ and $Vars_c$. The set *Fun* of function symbols of $\mathcal{L}$ is partitioned into Fun_u and Fun_c, where Fun_c is the set $\{+, \cdot\} \cup \mathbb{Q}$ and $\mathbb{Q}$ is the set of rational numbers. The set *Pred* of predicate symbols of $\mathcal{L}$ is partitioned into $Pred_u$ and $Pred_c$, that is the set $\{\geq, >, =\}$.

In order to distinguish terms representing rational numbers from all other terms, we assume that $\mathcal{L}$ is a typed language. This is necessary in order to give the semantics of formulas where both kinds of terms occur. We have two basic types: `tree`, that is, the type of finite trees, and `rat`, that

is, the type of rational numbers. Moreover we consider types constructed from basic types by using the type constructors $\times$ and $\rightarrow$. A variable $X \in \mathit{Vars}_u$ has type `tree`, and a variable $Y \in \mathit{Vars}_c$ has type `rat`. A predicate symbol and a function symbol in $\mathcal{L}$ have type, respectively, $\tau_1 \times \ldots \times \tau_n$ and $\tau_1 \times \ldots \times \tau_n \rightarrow \tau_{n+1}$, for some types $\tau_1,\ldots,\tau_n,\tau_{n+1} \in \{\texttt{tree},\texttt{rat}\}$. In particular, the predicate symbols $\geq$, $>$, and $=$ have type $\texttt{rat} \times \texttt{rat}$, the function symbols $+$ and $\cdot$ have type $\texttt{rat} \times \texttt{rat} \rightarrow \texttt{rat}$, and the rational numbers have type `rat`.

A term of type `rat` is a *linear polynomial* and has the following syntax

$$p ::= \ a \mid X \mid a{\cdot}X \mid p_1 + p_2$$

where a is a rational number and $X \in \mathit{Vars}_c$. Normally, we will write the term $a{\cdot}X$ as aX.

A term of type `tree` has the following syntax:

$$t ::= \ X \mid f(t_1,\ldots,t_n)$$

where $X \in \mathit{Vars}_u$, f is a function symbol of type $\tau_1 \times \ldots \times \tau_n \rightarrow \texttt{tree}$, and $t_1,\ldots,t_n$ are terms of type $\tau_1,\ldots,\tau_n$, respectively. A *term* is either a term of type `rat` or a term of type `tree`.

A *constraint* is a finite conjunction of *atomic constraints*, which are linear equations and inequations over rational numbers. Constraints have the following syntax:

$$c ::= \ \mathit{true} \mid \mathit{false} \mid p_1 \geq p_2 \mid p_1 > p_2 \mid p_1 = p_2 \mid c_1 \wedge c_2$$

The set of all constraints is denoted as $\mathit{LIN}_{\mathbb{Q}}$. A formula in the language $\mathcal{L}_c$ is either a constraint in $\mathit{LIN}_{\mathbb{Q}}$ or any formula constructed from a formula in $\mathcal{L}_c$ using the usual logical connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$, and the quantifiers $\exists$, $\forall$, such that the quantified variables are in the set Vars_c.

An *atom* is an atomic formula of the form $r(t_1,\ldots,t_n)$ where r is a predicate symbol not in $\{\geq,>,=\}$ of type $\tau_1 \times \ldots \times \tau_n$, and $t_1,\ldots,t_n$ are terms of type $\tau_1,\ldots,\tau_n$, respectively. A *literal* is either an atom or a negated atom. A *goal* is a conjunction $L_1 \wedge \ldots \wedge L_n$ of literals with $n \geq 0$. A *constrained goal* is a conjunction $c \wedge G$ where c is a constraint and G is a goal. A *clause* is a formula of the form $H \leftarrow c \wedge G$, where H is an atom and $c \wedge G$ is a constrained goal. A *constraint logic program* is a set of clauses.

A formula in the language $\mathcal{L}_u$ is either a formula in the language $\mathcal{L}_c$, or a literal or any formula constructed from a formula in $\mathcal{L}_u$ using the usual logical connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$, and $\leftrightarrow$, and the quantifiers $\exists$ and $\forall$.

If e is a term or a formula, then by $vars_u(e)$ and $vars_c(e)$ we denote, respectively, the set of variables of type `tree` and of type `rat` occurring in e. By $vars(e)$ we denote the set $vars_u(e) \cup vars_c(e)$ of variables. Finally, by $fvars(e)$ we denote the set of free variables of e. Given a constrained goal $c_1 \wedge G_1$ and a clause γ of the form $H \leftarrow c_1 \wedge c_2 \wedge G_1 \wedge G_2$, we call *local variables of* $c_1 \wedge G_1$ *in* γ those variables that occur in $c_1 \wedge G_1$ and not in the rest of γ. In the following, when dealing with local variables, we will omit to mention the reference to the clause γ when it is understood from the context. Given a clause γ: $H \leftarrow c \wedge G$, by $evars(\gamma)$ we denote the set $vars(c \wedge G) - vars(H)$ of the *existential variables* of γ. Given a set $X = \{X_1, \ldots, X_n\}$ of variables and a formula φ, by $\forall X\, \varphi$ we denote the formula $\forall X_1 \ldots \forall X_n\, \varphi$ and by $\exists X\, \varphi$ we denote the formula $\exists X_1 \ldots \exists X_n\, \varphi$. By $\forall(\varphi)$ we denote the *universal closure* of φ and by $\exists(\varphi)$ we denote the *existential closure* of φ.

In the following we will use the notion of substitution as defined in [43]. The substitution $\{X_1/t_1, \ldots, X_n/t_n\}$ is a *typed substitution* if, for all $i = 1, \ldots, n$, the type of X_i is equal to the type of t_i. The following lemma is important to establish which kind of substitutions are allowed to avoid the introduction of formulas that do not belong to the typed language $\mathcal{L}$.

Lemma 11 *Let φ be a formula in the language $\mathcal{L}_c$ or $\mathcal{L}_u$, if ϑ is a typed substitution, then $\varphi\vartheta$ is a formula in the language $\mathcal{L}_c$ or $\mathcal{L}_u$, respectively.*

Following the approach of [43], we provide a pre-interpretation J for the language $\mathcal{L}$ as follows. The pre-interpretation J assigns to the type `rat` the domain $\mathbb{Q}$ of rational numbers, and to the type `tree` the domain of ground terms constructed from the set of function symbols $Fun_u \cup \mathbb{Q}$. To the function symbols $+$ and $\cdot$, J assigns, respectively, the usual addition and multiplication operations in $\mathbb{Q}$. For the function symbols in $Fun_u \cup \mathbb{Q}$, J is defined as the usual Herbrand pre-interpretation.

The interpretation $\mathcal{Q}$ for the language $\mathcal{L}_c$ is based on the pre-interpretation J. $\mathcal{Q}$ assigns to the predicate symbols $\geq$, $>$, and $=$, respectively, the

usual ordering, strict ordering, and equality relations on $\mathbb{Q}$. For a formula φ of $\mathcal{L}_c$, the satisfaction relation $\mathcal{Q} \models \varphi$ is defined as usual in first order logic.

An interpretation I for the language $\mathcal{L}$ is a *$\mathcal{Q}$-interpretation* if (i) I is based on J, and (ii) for all formulas φ in $\mathcal{L}_c$, we have that $I \models \varphi$ iff $\mathcal{Q} \models \varphi$. For a given $\mathcal{Q}$-interpretation I and a clause $H \leftarrow c \wedge G$, we write $I \models H \leftarrow c \wedge G$ if we have that $I \models \forall(H \leftarrow c \wedge G)$, and we say that I is a *$\mathcal{Q}$-model* of the clause $H \leftarrow c \wedge G$. Also, I is a $\mathcal{Q}$-model of a program P if for all clauses γ in P we have $I \models \gamma$. Similarly to the case of logic programs, we can define *stratified* constraint logic programs and we can show that every stratified program P has a *perfect* $\mathcal{Q}$-model [29, 32, 45], denoted by $M(P)$.

We do not specify any particular method for solving constraints in $LIN_{\mathbb{Q}}$. We only assume that there exists a computable total function $solve : LIN_{\mathbb{Q}} \times \mathcal{P}_{fin}(Vars_c) \rightarrow LIN_{\mathbb{Q}}$, where $\mathcal{P}_{fin}(Vars_c)$ is the set of finite subsets of $Vars_c$, such that:

(i) for every constraint $c \in LIN_{\mathbb{Q}}$ and for every finite set of variables $X \subseteq Vars_c$, if $solve(c, X) = d$ then $\mathcal{Q} \models \forall X\, ((\exists Y\, c) \leftrightarrow d)$, where $Y = vars(c) - X$ and $vars(d) \subseteq X$;

(ii) $solve(c, \emptyset) = true$ iff c is satisfiable;

(iii) $solve(c, \emptyset) = false$ iff c is unsatisfiable.

In the following we also write $solve(c, X)$ as $c\,|_X$. A possible implementation of the *solve* function is based on the Fourier-Motzkin variable elimination algorithm [11].

A clause γ: $H \leftarrow c \wedge G$ is in *normal form* if the only terms of type `rat` occurring in G are distinct variables not occurring in H, and c has no local variables in γ. Given a clause γ_1, it is always possible to transform γ_1 into a clause γ_2 in normal form, such that γ_1 and γ_2 have the same $\mathcal{Q}$-models. Indeed, we now introduce the Normalization Algorithm which transforms a given clause γ_1 into a clause γ_2 in normal form. The clause γ_2 is called *a normal form of* γ_1. In the following algorithm, by $G[t]$ we denote a goal where the term t occurs.

Normalization Algorithm

Input: a clause $\gamma_1 : H \leftarrow c_1 \wedge G_1$;

Output: a clause $\gamma_2 : H \leftarrow c_2 \wedge G_2$ such that γ_2 is in normal form and γ_2 has the same $\mathcal{Q}$-models of γ_1.

Step 1. Start from the constrained goal $c_1 \wedge G_1$ and apply the following rewriting rule until no further application is possible:

$$c \wedge G[u] \Longrightarrow c \wedge V = u \wedge G[V]$$

where u is either a non-variable term of type `rat`, or a variable of type `rat` which occurs in H or has multiple occurrences in $G[t]$, and V is a variable of type `rat` that does not occur in $\{H, c, G[t]\}$.

Step 2. Let the constrained goal $c'_1 \wedge G'_1$ be the result of Step 1. If c'_1 has no local variables in the clause $H \leftarrow c'_1 \wedge G'_1$ then let the constraint c_2 be c'_1 else let c_2 be the constraint $(c'_1)|_X$, where X is the set $vars_c(c'_1) - (vars_c(H) \cup vars_c(G'_1))$ of the local variables of c'_1. Let G_2 be the goal G'_1 and return the clause $H \leftarrow c_2 \wedge G_2$.

The next two theorems show that the Normalization Algorithm is correct and terminates for every input clause.

Theorem 10 *Let the clause γ_1 be the input of the Normalization Algorithm and let γ_2 be the output, then γ_2 is a clause, γ_2 is in normal form, and for every $\mathcal{Q}$-interpretation I, $I \models \gamma_1$ iff $I \models \gamma_2$.*

Proof: Let γ_1 be $H \leftarrow c_1 \wedge G_1$. The output γ_2 of the Normalization Algorithm is a clause of the typed language $\mathcal{L}$ because, by hypothesis, γ_1 is a clause of $\mathcal{L}$, we substitute every term of type `rat` occurring in the goal G_1 for a fresh new variable of type `rat` and we add a constraint which is an equality between terms of type `rat`. By construction, γ_2 is in normal form because every outermost occurrence of a linear polynomial in G_2 is a distinct variable of type `rat` that may occur in c_2, but not in H, and

the local variables of c_2 have been eliminated by using the function *solve*. The soundness of Step 1 of the algorithm is motivated by the following axiom $\forall(\varphi[t] \leftrightarrow \exists Y(Y = t \wedge \varphi[Y]))$ of the first order predicate calculus with equality [49], where φ is any first order formula, t is free for Y in φ, and Y does not occur in t. These conditions are satisfied because at Step 1 we test that each variable V introduced during the rewriting process does not occur in the clause obtained so far. Finally, by definition of the *solve* function, the clauses γ_1 and γ_2 have the same $\mathcal{Q}$-models. □

Theorem 11 *The Normalization Algorithm terminates for every input clause.*

Proof: Let the clause $\gamma : H \leftarrow c \wedge G$ be the input of the Normalization Algorithm. The number of terms occurring in G that are either a non-variable term of type `rat`, or a variable of type `rat` which occurs in H or has multiple occurrences in $G[t]$ is finite. An application of the rewriting rule at Step 1 replaces an occurrence of a term of type `rat` in G by a fresh new variable. Therefore we can infer that each application of the rewriting rule strictly decreases the number of terms occurring in G that are either a non-variable term of type `rat`, or a variable of type `rat` which occurs in H or has multiple occurrences in $G[t]$. As a consequence, Step 1 terminates after a finite number of applications of the rewriting rule. The termination of Step 2 is guaranteed by the definition of the *solve* function. □

Let γ_1 and γ_2 be two clauses, we write $\gamma_1 \cong \gamma_2$ if the clause $H \leftarrow c_1 \wedge B_1$ is a normal form of γ_1, the clause $H \leftarrow c_2 \wedge B_2$ is a normal form of γ_2, and exists a variable renaming ρ such that: (i) $x\rho$ is x for all $x \in vars(H)$, (ii) $B_1 =_{AC} B_2\rho$, and (iii) $\mathcal{Q} \models \forall\ (c_1 \leftrightarrow c_2\rho)$, where by $=_{AC}$ we mean equality modulo the equational theory of associativity and commutativity of conjunction. We refer to this theory as the $AC_\wedge$ *theory* [81]. The equivalence relation can be extended to sets of clauses as follows: let $\{\gamma_1, \ldots, \gamma_m\}$ and $\{\delta_1, \ldots, \delta_n\}$ be two sets of clauses, then $\{\gamma_1, \ldots, \gamma_m\} \cong \{\delta_1, \ldots, \delta_n\}$ if, for $i = 1, \ldots, m$ and $j = 1, \ldots, n$, $H \leftarrow c_i \wedge B_1$ is a normal form of γ_i and $H \leftarrow d_j \wedge B_2$ is a normal form of δ_j, and exists a variable renaming ρ such that: (i) $x\rho$ is x for all $x \in vars(H)$, (ii) $B_1 =_{AC} B_2\rho$, and (iii) $\mathcal{Q} \models \forall\ ((c_1 \vee \ldots \vee c_m) \leftrightarrow (d_1 \vee \ldots \vee d_n)\rho)$.

The following Lemmas 12-15 state some properties of the $\cong$ relation which will be used in the following.

Lemma 12 $\cong$ *is an equivalence relation.*

Proof: It can be easily seen that $\cong$ is reflexive. The symmetry of $\cong$ is a consequence of the fact that every variable renaming ρ is invertible. Let us now prove that $\cong$ is transitive. Assume that $\gamma_1 \cong \gamma_2$ and $\gamma_2 \cong \gamma_3$. By definition of $\cong$, there exist normal forms $H \leftarrow c_1 \wedge B_1$, $H \leftarrow c_2 \wedge B_2$, and $H \leftarrow c_3 \wedge B_3$ of γ_1, γ_2, and γ_3, respectively, and there exist two variable renamings ρ_1 and ρ_2 such that: (i) for all $x \in vars(H)$, $x\rho_1$ and $x\rho_2$ are both equal to x, (ii) $B_1 =_{AC} B_2\rho_1$ and $B_2 =_{AC} B_3\rho_2$, and (iii) $\mathcal{Q} \models \forall\,(c_1 \leftrightarrow c_2\rho_1)$ and $\mathcal{Q} \models \forall\,(c_2 \leftrightarrow c_3\rho_2)$. As a consequence, we have that, for all $x \in vars(H)$, $x\rho_2\rho_1$ is x, $B_1 =_{AC} B_3\rho_2\rho_1$, and $\mathcal{Q} \models \forall\,(c_1 \leftrightarrow c_3\rho_2\rho_1)$. Thus, $\gamma_1 \cong \gamma_3$. □

Lemma 13 *If* $\gamma_1 \cong \gamma_2$ *then, for every* $\mathcal{Q}$*-interpretation* I, $I \models \gamma_1$ *iff* $I \models \gamma_2$.

Proof: Assume $\gamma_1 \cong \gamma_2$, and let $H \leftarrow c_1 \wedge B_1$ and $H \leftarrow c_2 \wedge B_2$ be normal forms of γ_1 and γ_2, respectively. Then $B_1 =_{AC} B_2\rho$ and $\mathcal{Q} \models \forall\,(c_1 \leftrightarrow c_2\rho)$, for some variable renaming ρ. Recalling that the relation $I \models \varphi$ is invariant under variable renaming of φ, we have that for every $\mathcal{Q}$-interpretation I, $I \models B_1$ iff $I \models B_2$ and $I \models c_1$ iff $I \models c_2$, and thus we get the thesis. □

Lemma 14 *Let* γ_1 *be the input of the Normalization Algorithm and* γ_2 *the output, then* $\gamma_1 \cong \gamma_2$.

Proof: Let γ_1 be $H \leftarrow c_1 \wedge B_1$ and γ_2 be $H \leftarrow c_2 \wedge B_2$. Also, let $\gamma_3 : H \leftarrow c_3 \wedge B_3$ be a normal form of γ_1 (γ_2 is already in normal form). By construction, the goals B_3 and B_2 differ only on the terms of type `rat`, which are distinct variables. Therefore, there exists a variable renaming ρ such that $B_3 =_{AC} B_2\rho$. Now we have to show that $\mathcal{Q} \models \forall\,(c_3 \leftrightarrow c_2\rho)$, where by construction the constraint c_3 is equivalent to $\exists W_1\,(c_1 \wedge eq_{B_1})$ and the constraint c_2 is equivalent to $\exists W_2\,(c_1 \wedge eq'_{B_1})$. In these formulas, W_1 and W_2 are the sets of local variables of $c_1 \wedge eq_{B_1}$ and $c_1 \wedge eq'_{B_1}$ in

the clauses γ_3 and γ_2, respectively, and eq_{B_1} and eq'_{B_1} are the equalities introduced by the Normalization Algorithm. Note that ρ acts as the identity substitution over the sets of variables W_1 and W_2. By construction we have that $\mathcal{Q} \models \forall\ (eq_{B_1} \leftrightarrow eq'_{B_1}\rho)$ and the set W_1 is equal to the set W_2. As a consequence, $\mathcal{Q} \models \forall\ (\exists W_2\,(c \wedge eq_{B_1}) \leftrightarrow (\exists W_1\,(c \wedge eq'_{B_1}))\rho)$. □

Lemma 15 *Let* Γ_1 *be the set* $\{H \leftarrow c_1 \wedge B, \ldots, H \leftarrow c_m \wedge B\}$ *of clauses and* Γ_2 *be the set* $\{H \leftarrow d_1 \wedge B, \ldots, H \leftarrow d_n \wedge B\}$ *of clauses,* $\Gamma_1 \cong \Gamma_2$ *iff* $\mathcal{Q} \models \forall\,((\exists U_1\,c_1) \vee \ldots \vee (\exists U_m\,c_m) \leftrightarrow (\exists V_1\,d_1) \vee \ldots \vee (\exists V_n\,d_n))$ *where, for* $i = 1, \ldots, m$, U_i *is the set of local variables of* c_i *in* $H \leftarrow c_i \wedge B$ *and, for* $j = 1, \ldots, n$, V_j *is the set of local variables of* d_j *in* $H \leftarrow d_j \wedge B$.

Proof: For reasons of simplicity we now prove the lemma in the case where Γ_1 is the singleton $\{H \leftarrow c \wedge B\}$ and Γ_1 is the singleton $\{H \leftarrow d \wedge B\}$. The proof for the general case is a straightforward generalization of the following proof. Assume that $H \leftarrow c \wedge B \cong H \leftarrow d \wedge B$. We want to show that $\mathcal{Q} \models \forall\,(\exists U\,c \leftrightarrow \exists V\,d)$. Take a variable assignment σ and a $\mathcal{Q}$-interpretation I such that $I, \sigma \not\models H$ and $I, \sigma \models B$. By the assumption and by Lemma 13, we have that $I, \sigma \models H \leftarrow (\exists U c) \wedge B$ iff $I, \sigma \models H \leftarrow (\exists V d) \wedge B$. Hence, $I, \sigma \models \exists U c$ iff $I, \sigma \models \exists V d$. By definition of the $\mathcal{Q}$-interpretation, we have that $\mathcal{Q}, \sigma \models \exists U c$ iff $\mathcal{Q}, \sigma \models \exists V d$. Thus, by the definition of the $\models$ relation, we have $\mathcal{Q} \models \forall\,(\exists U\,c \leftrightarrow \exists V\,d)$.

Now, assume that $\mathcal{Q} \models \forall\,(\exists U\,c \leftrightarrow \exists V\,d)$. There exist two clauses $H \leftarrow c' \wedge B'$ and $H \leftarrow d' \wedge B'$ that are normal forms of $H \leftarrow c \wedge B$ and $H \leftarrow d \wedge B$, respectively, such that c' is equivalent to $\exists W_1\,(c \wedge eq_B)$ and d' is equivalent to $\exists W_2\,(d \wedge eq_B)$, where the formula eq_B is the conjunction of the equalities introduced by the Normalization Algorithm, and W_1 and W_2 are the local variables of $c \wedge eq_B$ and $d \wedge eq_B$ in $H \leftarrow c \wedge eq_B \wedge B'$ and $H \leftarrow d \wedge eq_B \wedge B'$, respectively. Let W be the set $W_1 \cup W_2$ and observe that c' is equivalent to $\exists W((\exists U c) \wedge eq_B)$ and d' is equivalent to $\exists W((\exists V d) \wedge eq_B)$. The first equivalence holds because the variables in U do not occur in eq_B, therefore we can distribute the existential quantification over the conjunction, and by construction the variables in W_2 either belong to W_1 or do not occur in $c \wedge eq_B$. Symmetrically, a similar observation holds for the other equivalence. By using the assumption, we can infer

that both c' and d' are equivalent to $\exists W\,((\exists U c) \wedge eq_B)$. As a consequence, $\mathcal{Q} \models \forall\,(c \leftrightarrow d)$ and $H \leftarrow c \wedge B \cong H \leftarrow d \wedge B$. □

Since every clause can be transformed into an equivalent clause in normal form, when presenting the folding rules and the corresponding algorithms, without loss of generality we will assume that they apply to clauses in normal form.

4.2 The Standard Folding Rule

In the literature there are several slightly different versions of the folding transformation rule for constraint logic programs [9, 24, 29, 45]. These versions differ on the applicability conditions and on the number of clauses that can be folded in a single rule application. In this section we introduce the rule **Fold$_1$**, whose applicability conditions are the ones given in [29]. For reasons of simplicity, in the following presentation, in contrast to [29], our rule allows the folding of at most one clause at a time (like in [9, 24]). The general case, where $n \geq 1$ clauses can be folded at once, can be addressed as a straightforward generalization of the methods we will present in the sequel.

Definition 7 (Rule Fold$_1$)
Let γ and δ be clauses of the form

γ: $H \leftarrow c \wedge G$
δ: $K \leftarrow d \wedge B$

such that γ and δ are in normal form and have no common variables. Suppose that there exist a constraint e, a substitution ϑ, and a goal R such that:
1) $H \leftarrow c \wedge G \;\cong\; H \leftarrow e \wedge d\vartheta \wedge B\vartheta \wedge R$;
2) for every variable X in $evars(\delta)$, the following conditions hold: (i) $X\vartheta$ is a variable not occurring in $\{H, e, R\}$, and (ii) $X\vartheta$ does not occur in the term $Y\vartheta$, for every variable Y occurring in $d \wedge B$ and different from X.
By *folding clause* γ *using clause* δ we derive the clause η: $H \leftarrow e \wedge K\vartheta \wedge R$.

Now we present a correctness result for the folding rule **Fold$_1$** with respect to the perfect model semantics. This result follows immediately from [9, 24, 29].

A *transformation sequence* is a sequence $P_0, \ldots, P_n$ of programs such that, for $k = 0, \ldots, n-1$, program P_{k+1} is derived from program P_k by an application of one of the following transformation rules: *definition*, *unfolding*, *folding* (Here we refer to the definition rule and the unfolding rule as they presented in [29]). By $Defs_n$ we denote the set of clauses introduced by the definition rule during the construction of $P_0, \ldots, P_n$.

Program P_{k+1} is derived from program P_k by an application of the folding rule **Fold$_1$** as follows. Suppose that γ is a clause in P_k, δ is a clause in $Defs_n$, and η is the clause derived by folding γ using δ as described in Definition 7. Then $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta\}$.

Theorem 12 [29] *Let P_0 be a stratified program and let $P_0, \ldots, P_n$ be a transformation sequence. Suppose that, for $k = 0, \ldots, n-1$, if P_{k+1} is derived from P_k by folding clause γ using clause δ, then there exists j, with $0 < j < n$, such that $\delta \in P_j$ and program P_{j+1} is derived from P_j by unfolding δ w.r.t. a positive literal in its body. Then $P_0 \cup Defs_n$ and P_n are stratified and $M(P_0 \cup Defs_n) = M(P_n)$.*

In the rest of this section, we propose an algorithm to determine whether or not a clause γ can be folded using a clause δ, according to Definition 7. The objective of our folding algorithm is to find ϑ, e, and R that satisfy the equivalence $H \leftarrow c \wedge G \ \cong \ H \leftarrow e \wedge d\vartheta \wedge B\vartheta \wedge R$ (see point (1) of Definition 7). Basically, this can be viewed as solving a *matching* problem [81], stated by the clause equivalence, with the extra conditions of Point (2) of Definition 7. The key idea of our algorithm consists in solving this matching problem in two steps. In the first step we apply a *goal matching procedure* which performs a matching between goals that is a specialized form of matching modulo the $AC_\wedge$ theory [81]. In the second step we apply a *constraint matching procedure* which performs a matching of constraints modulo the theory of constraints.

The output of the algorithm is either a clause η, derived by folding γ using δ, or FAILURE. In the former case the algorithm computes a sub-

stitution ϑ, a constraint e, and a goal R such that the Conditions (1) and (2) of Definition 7 are satisfied. Since Definition 7 does not determine in a unique way ϑ, e, and R, our folding algorithm is nondeterministic and in different runs it may compute different output clauses.

Let us now present the goal matching procedure which, given two clauses $\gamma\colon H \leftarrow c \wedge G$ and $\delta\colon K \leftarrow d \wedge B$ as input, either computes, if at all possible, two clauses $\gamma'\colon H \leftarrow c \wedge B\vartheta_1 \wedge R$ and $\delta'\colon K\vartheta_1 \leftarrow d\vartheta_1 \wedge B\vartheta_1$ such that G is equal to $B\vartheta_1 \wedge R$ modulo the $\mathrm{AC}_\wedge$ theory, for some substitution ϑ_1 and goal R, or returns FAILURE, otherwise. As already mentioned, the problem to be solved in the goal matching procedure can be viewed as the problem of matching B against G, modulo the $\mathrm{AC}_\wedge$ theory, with the additional conditions on the matching substitution regarding the existential variables of δ (see Condition (2) of Definition 7). This motivates the introduction of an *ad hoc* algorithm, inspired to the standard matching techniques, but tailored to our application.

Goal Matching Procedure: **GM**

Input: two clauses $\gamma\colon H \leftarrow c \wedge G$ and $\delta\colon K \leftarrow d \wedge B$ in normal form without common variables.
Output: either (a) two clauses $\gamma'\colon H \leftarrow c \wedge B\vartheta_1 \wedge R$ and $\delta'\colon K\vartheta_1 \leftarrow d\vartheta_1 \wedge B\vartheta_1$ in normal form, such that: (1) $\gamma \cong \gamma'$ and (2) for every variable X in $evars(\delta)$, the following conditions hold: (2.i) $X\vartheta_1$ is a variable not occurring in $\{H, R\}$, and (2.ii) $X\vartheta_1$ does not occur in the term $Y\vartheta_1$, for every variable Y occurring in $d \wedge B$ and different from X, if possible, or (b) FAILURE.

Step 1. Compute an injective mapping μ from the set $\overline{B}$ of the literals occurring in B, to the set $\overline{G}$ of the literals occurring in G such that for all $L \in \overline{B}$, L and $\mu(L)$ are either both positive or both negative literals, and have the same predicate symbol with the same arity. If there exists no such injective mapping μ then return FAILURE.

Step 2. Consider the set of equations $S = \{L = \mu(L) \mid L \in \overline{B}\}$. Apply as long as possible the following rewriting rules to the set S:

(i) $\{\neg A_1 = \neg A_2\} \cup S \Rightarrow \{A_1 = A_2\} \cup S$;
(ii) $\{a(s_1, \ldots, s_n) = a(t_1, \ldots, t_n)\} \cup S \Rightarrow \{s_1 = t_1, \ldots, s_n = t_n\} \cup S$;
(iii) $\{a(s_1, \ldots, s_m) = b(t_1, \ldots, t_n)\} \cup S \Rightarrow$ **fail**, if a is different from b;
(iv) $\{a(s_1, \ldots, s_n) = X\} \cup S \Rightarrow$ **fail**, if X is a variable;
(v) $\{X = t_1\} \cup S \Rightarrow$ **fail**, if X is a variable and there is an equation $X = t_2$ in S such that t_1 is different from t_2.

Step 3. Let R be the conjunction of all literals in $\overline{G} - \{\mu(L) \mid L \in \overline{B}\}$. IF S is not rewritten into **fail** and for every equation $X = t$ in S such that $X \in evars(\delta)$ we have that: (i) the term t is a variable not occurring in $\{H, R\}$ and (ii) there is no $Y \in vars(d \wedge B)$ different from X such that $Y = r$ is an equation in S and $t \in vars(r)$ THEN return the clauses $\gamma' : H \leftarrow c \wedge B\vartheta_1 \wedge R$ and $\delta' : K\vartheta_1 \leftarrow d\vartheta_1 \wedge B\vartheta_1$, where ϑ_1 is the set $\{X/t \mid X = t \in S\}$ ELSE go to Step 1 looking for a new injective mapping μ.

The goal matching procedure **GM** is nondeterministic because, in general, more than one mapping μ may lead to the construction of a matching substitution ϑ_1 and different mappings μ may determine different pairs of output clauses. At Step 1 **GM** looks for a mapping μ between literals and then at Step 3 it tests whether or not that mapping μ allows the construction of the matching substitution ϑ_1. For the sake of simplicity, we presented the Step 1 of the goal matching procedure **GM** in a generate-and-test style, without taking into account the efficiency aspects. However, one can show that it is possible to generate the injective mapping μ nondeterministically in polynomial time. This result cannot be improved, since the problem of matching modulo the equational theory $\mathrm{AC}_\wedge$ has been shown NP-complete [8]. Finally, note that since the input clauses γ and δ are in normal form, the substitution ϑ_1 computed by the goal matching procedure **GM** is a renaming substitution when restricted to the variables of type `rat`.

The following two theorems state that the goal matching procedure **GM** is sound and complete, that is, it finds the suitable matching substitution ϑ_1 and goal R if and only if they exist.

Theorem 13 (Soundness of the Goal Matching Procedure) *Let the clauses* $\gamma\colon H \leftarrow c \wedge G$ *and* $\delta\colon K \leftarrow d \wedge B$ *be the input of the goal matching procedure* **GM**. *If the output of* **GM** *is the pair of clauses* $\gamma'\colon H \leftarrow c \wedge B\vartheta_1 \wedge R$ *and* $\delta'\colon K\vartheta_1 \leftarrow d\vartheta_1 \wedge B\vartheta_1$, *then*

1. γ' *and* δ' *are in normal form,*
2. $\gamma \cong \gamma'$, *and*
3. *the substitution* ϑ_1 *is such that, for every variable* X *in* $evars(\delta)$, *the following conditions hold: (i)* $X\vartheta_1$ *is a variable not occurring in* $\{H, R\}$, *and (ii)* $X\vartheta_1$ *does not occur in the term* $Y\vartheta_1$, *for every variable* Y *occurring in* $d \wedge B$ *and different from* X.

Proof: (*1.*) We show that $\gamma'\colon H \leftarrow c \wedge B\vartheta_1 \wedge R$ and $\delta'\colon K\vartheta_1 \leftarrow d\vartheta_1 \wedge B\vartheta_1$ are clauses of the typed first order language $\mathcal{L}$. By Lemma 11, it is enough to show that the substitution ϑ_1 is a typed substitution, and that R is a goal. According to the definition of $\mathcal{L}$ provided in Section 4.1, every predicate of $\mathcal{L}$ has a unique type. Therefore, when at Step 2 we apply rules (i) and (ii), we generate equations between terms of the same type. As a consequence, the substitution ϑ_1 computed at the end of Step 3 is a typed substitution. By Lemma 11, $B\vartheta_1$ is a goal in $\mathcal{L}$. Moreover, also R is a goal in $\mathcal{L}$, because it is constructed as a conjunction of literals in $\mathcal{L}$. Now we show that γ' and δ' are in normal form. In order to prove this, we show that the substitution ϑ_1 is a variable renaming when restricted to the variables of type `rat` occurring in B. Assume that the rewriting process of Step 2 terminates and does not return **fail**. Hence, by rules (i)-(v), S is a set of equations of the form $X = t$. Furthermore, assume that the IF test performed at Step 3 succeeds, then ϑ_1 is a substitution satisfying the following properties: (i) every existential variable X of δ is bound to a variable and (ii) there is no variable Y different from X that it is bound to a term in which $X\vartheta_1$ occurs. As a consequence, ϑ_1 is a variable renaming when restricted to the existential variables of δ. By hypothesis, the clauses γ and δ are in normal form, and therefore the variables of type `rat` occurring in B are distinct and existential variables. Thus, by

construction, the computed substitution ϑ_1 is a variable renaming when restricted to the variables of type `rat` and the clauses $H \leftarrow c \wedge B\vartheta_1 \wedge R$ and $K\vartheta_1 \leftarrow d\vartheta_1 \wedge B\vartheta_1$ are in normal form.

(*2.*) $H \leftarrow c \wedge G \cong H \leftarrow c \wedge B\vartheta_1 \wedge R$ holds under the empty variable renaming if $G =_{AC} B\vartheta_1 \wedge R$ because the clauses are in normal form. Step 2 implements a standard matching algorithm modulo the empty equational theory [81]. By Step 1 we consider G and B as sets of literals and nondeterministically compute an injective mapping from the literals in B to those in G. In this way we are exploiting the associativity and commutativity of $\wedge$. Finally, by Step 3 we compute the substitution ϑ_1 from the set S of equations. If these steps terminate by computing a substitution ϑ_1 then $G =_{AC} B\vartheta_1 \wedge R$.

(*3.*) By Step 3 we are ensured that ϑ_1 satisfies Condition (3) of the theorem. □

Theorem 14 (Completeness of the Goal Matching Procedure) *The goal matching procedure* **GM** *terminates for all input clauses. Moreover, if the input clauses are* $\gamma\colon H \leftarrow c \wedge G$ *and* $\delta\colon K \leftarrow d \wedge B$*, where* γ *and* δ *are in normal form and do not have common variables, and there exist a substitution* ϑ_1 *and a goal* R *such that:*

1. $H \leftarrow c \wedge G \cong H \leftarrow c \wedge B\vartheta_1 \wedge R$ *and*
2. *for every variable* X *in* $evars(\delta)$*, the following conditions hold: (i)* $X\vartheta_1$ *is a variable not occurring in* $\{H, R\}$*, and (ii)* $X\vartheta_1$ *does not occur in the term* $Y\vartheta_1$*, for every variable* Y *occurring in* $d \wedge B$ *and different from* X*,*

then the output of **GM** *is not FAILURE.*

Proof: The termination of Steps 1 and 3 is straightforward. The termination of Step 2 is motivated by the following observations. The rules (i) and (ii) can be applied a finite number of times, for any set S of equations, because in the application of each rule the number of function and predicate symbols in S strictly decreases (here we are not counting the occurrences of the equality symbol $=$). Rules (iii), (iv) and (v) can be applied

at most once, because no rule can be applied to **fail**. If the condition of the *IF-THEN-ELSE* at Step 3 fails, then the algorithm backtracks to Step 1 to compute a different mapping. Since, there are finitely many possible injective mappings from the finite set $\overline{B}$ to the finite set $\overline{G}$, backtracking can be performed a finite number of times.

Now, assume that it is possible to find a substitution ϑ_1 and a goal R such that $H \leftarrow c \wedge G \cong H \leftarrow c \wedge B\vartheta_1 \wedge R$ and Condition (2) of the theorem holds. By the definition of equivalence, $H \leftarrow c \wedge G \cong H \leftarrow c \wedge B\vartheta_1 \wedge R$ iff $G =_{AC} B\vartheta_1 \wedge R$. Let us now consider two goals Q_1 and Q_2: there exists a goal Q_3 such that $Q_1 =_{AC} Q_2 \wedge Q_3$ iff it is possible to find an injective mapping μ from the literals in Q_2 to the literals in Q_1. There are finitely many possible mappings because Q_1, Q_2, and Q_3 are finite conjunctions. In our case, Q_1, Q_2, and Q_3 are G, $B\vartheta_1$, and R, respectively. By backtracking, Step 1 computes all possible injective mappings from $\overline{B}$ to $\overline{G}$ until it is possible to compute, at Steps 2 and 3, a substitution ϑ_1 such that $\mu(L_1) \wedge \ldots \wedge \mu(L_n) = (L_1 \wedge \ldots \wedge L_n)\vartheta_1$, where the goal $L_1 \wedge \ldots \wedge L_n$ is B. Note that here we have used the equality '=', by which here we denote syntactic equality, instead of the equality modulo associativity and commutativity of conjunction '$=_{AC}$'. This is because, we exploit associativity and commutativity of conjunction by choosing the injective mapping μ at Step 1. Step 2 implements a standard complete matching algorithm [81]. By using these observations and the completeness of Step 2, we have that if it is possible to find a substitution ϑ_1 such that $G =_{AC} B\vartheta_1 \wedge R$ and Condition (2) of the theorem holds, then the goal matching procedure does not return FAILURE, but computes the clauses γ' and δ'. □

The constraint matching procedure takes as input two clauses γ' and δ' which are an output of the goal matching procedure. When presenting the constraint matching procedure we will assume that they are of the form: $\gamma' : H \leftarrow c \wedge B \wedge R$ and $\delta' : K \leftarrow d \wedge B$. The constraint matching procedure either computes a clause $\gamma'' : H \leftarrow e \wedge d \wedge B \wedge R$, such that $H \leftarrow c \wedge B \wedge R \cong H \leftarrow e \wedge d \wedge B \wedge R$ and $evars(\delta') \cap vars(e) = \emptyset$ (see Condition 2.i of Definition 7), or it returns FAILURE. Note that, by Lemma 15, the

equivalence $H \leftarrow c \wedge B \wedge R \cong H \leftarrow e \wedge d \wedge B \wedge R$ holds iff $\mathcal{Q} \models \forall(c \leftrightarrow \exists W(e \wedge d))$, where W is the set of the local variables of $e \wedge d$. By these requirements on the constraint e, and by observing that the local variables of e in γ'' can always be eliminated, without loss of generality we can assume that $vars(e) = (vars(c) \cup vars(d)) - (vars(c) \cap vars(d))$. Let us now consider the formula $\hat{e}$, defined as follows.

Definition 8 Let $\hat{e}$ denote the formula $\forall Z\,(d \rightarrow c)$, where c and d are the constraints occurring in the input clauses γ' and δ', respectively, which is the output of the **GM** procedure and $Z = vars(c) \cap vars(d)$.

From Definition 8, we immediately get that $fvars(\hat{e}) = (vars(c) \cup vars(d)) - Z$. In the following Lemmas 16, 17, and 18, we are going to show that, if there exists a formula e' in $\mathcal{L}_c$ such that $\mathcal{Q} \models \forall(c \leftrightarrow \exists W'(e' \wedge d))$, where W' is the set of the local variables of $e' \wedge d$, then $\hat{e}$ is, indeed, the most general such e', in the sense that $\mathcal{Q} \models \forall(e' \rightarrow \hat{e})$.

Lemma 16 *If γ' is the clause $H \leftarrow c \wedge B \wedge R$ in normal form and δ' is the clause $K \leftarrow d \wedge B$ in normal form then $\mathcal{Q} \models \forall(\exists Y(\hat{e} \wedge d) \rightarrow c)$, where $Y = fvars(\hat{e} \wedge d) - vars(\{H, B \wedge R\})$.*

Proof: Let σ be any variable assignment. Assume that $\mathcal{Q}, \sigma \models \exists Y(\,\forall Z\,(d \rightarrow c) \wedge d)$. Then, by eliminating the universal quantifier, we have $\mathcal{Q}, \sigma \models \exists Y(d \rightarrow c \wedge d)$ and, hence, by Modus Ponens, $\mathcal{Q}, \sigma \models \exists Y\, c$. Now, since $Y \cap vars(c) = \emptyset$, we have that $\mathcal{Q}, \sigma \models c$. Thus, recalling that $\forall Z\,(d \rightarrow c)$ is the definition of $\hat{e}$, we have proved that $\mathcal{Q} \models \forall(\exists Y(\hat{e} \wedge d) \rightarrow c)$. □

Lemma 17 *If γ' is the clause $\leftarrow c \wedge B \wedge R$ in normal form and δ' is the clause $K \leftarrow d \wedge B$ in normal form then for all formulas e' in $\mathcal{L}_c$ such that $fvars(e') = (vars(c) \cup vars(d)) - Z$, if $\mathcal{Q} \models \forall(\exists W'(e' \wedge d) \rightarrow c)$, where $W' = fvars(e' \wedge d) - vars(\{H, B \wedge R\})$, then $\mathcal{Q} \models \forall(e' \rightarrow \hat{e})$.*

Proof: Let e' be a formula in $\mathcal{L}_c$ such that $fvars(e') = (vars(c) \cup vars(d)) - Z$ and $\mathcal{Q} \models \forall(\exists W'(e' \wedge d) \rightarrow c)$. Since $vars(c) \subseteq vars(\{H, B \wedge R\})$, then the variables in W' do not occur in c, thus we have that $\mathcal{Q} \models \forall((e' \wedge d) \rightarrow c)$.

Then, by applying suitable logical equivalences, we get $\mathcal{Q} \models \forall(e' \rightarrow (d \rightarrow c))$. By hypothesis, the variables Z do not occur in e', and therefore we can move the quantifier on Z inward and obtain $\mathcal{Q} \models \forall(e' \rightarrow \forall Z(d \rightarrow c))$. Finally, the lemma follows recalling that $\forall Z(d \rightarrow c)$ is the definition of $\hat{e}$.□

Lemma 18 *If γ' is the clause $H \leftarrow c \wedge B \wedge R$ in normal form and δ' is the clause $K \leftarrow d \wedge B$ in normal form then for all formulas e' in $\mathcal{L}_c$ such that $fvars(e') = (vars(c) \cup vars(d)) - Z$, if $\mathcal{Q} \models \forall(c \leftrightarrow \exists W'(e' \wedge d))$, where $W' = fvars(e' \wedge d) - vars(\{H, B \wedge R\})$, then $\mathcal{Q} \models \forall(c \leftrightarrow \exists Y(\hat{e} \wedge d))$, where $Y = fvars(\hat{e} \wedge d) - vars(\{H, B \wedge R\})$.*

Proof: By Lemma 16 we have that $\mathcal{Q} \models \forall(\exists Y(\hat{e} \wedge d) \rightarrow c)$. Let us assume that $\mathcal{Q} \models \forall(\exists W'(e' \wedge d) \rightarrow c)$, then, by Lemma 17, we have $\mathcal{Q} \models \forall(e' \rightarrow \hat{e})$. Let us now assume that $\mathcal{Q} \models \forall(c \rightarrow \exists W'(e' \wedge d))$. As a consequence, recalling that by hypothesis the sets W and Y are equal, we have that $\mathcal{Q} \models \forall(c \rightarrow \exists Y(\hat{e} \wedge d))$. □

Note that, in general, the formula $\hat{e}$ is not a constraint and therefore the constraint matching procedure cannot return $H \leftarrow \hat{e} \wedge d \wedge B \wedge R$. Thus, starting from the formula $\hat{e}$, we need to construct a constraint e such that $\mathcal{Q} \models \forall(\exists W(e \wedge d) \leftrightarrow \exists Y(\hat{e} \wedge d))$. (One can show that the set W is equal to the set Y.) In order to find such e, we recall that it is always possible to rewrite the formula $\hat{e}$ into a logically equivalent finite disjunction $e_1 \vee \ldots \vee e_n$ of constraints such that $fvars(\hat{e}) = vars(e_1 \vee \ldots \vee e_n)$. Thus, we have the following property, whose proof is straightforward.

Lemma 19 *If $\hat{e}$ is equivalent to $e_1 \vee \ldots \vee e_n$, for some constraints $e_1, \ldots, e_n$, then $\mathcal{Q} \models \forall(\exists Y(e_i \wedge d) \rightarrow c)$, where $Y = fvars(\hat{e} \wedge d) - vars(\{H, B \wedge R\})$, for $i = 1, \ldots, n$.*

Now we present the constraint matching procedure **CM$_1$** which is a nondeterministic procedure whose soundness follows from Lemmas 16, 17, 18, and 19. At Step 1 of the **CM$_1$** procedure the formula $\widehat{e}$ is rewritten into a finite disjunction $e_1 \vee \ldots \vee e_n$ of constraints. At Step 2, the procedure tests whether or not there exists a constraint e_i such that $\mathcal{Q} \models \forall(c \rightarrow \exists Y\,(e_i \wedge d))$ and, thus, by Lemma 19, it can be taken to be the constraint e we want

to construct. Note that this technique for finding e is very simple but, unfortunately, it leads to the incompleteness of the constraint matching procedure $\mathbf{CM_1}$ because, in general, there may be a constraint e such that $\mathcal{Q} \models \forall(c \leftrightarrow \exists Y\,(e \wedge d))$, and yet, $\mathcal{Q} \not\models \forall(c \rightarrow \exists Y\,(e_i \wedge d))$ for any e_i.

Constraint Matching Procedure: $\mathbf{CM_1}$

Input: two clauses $\gamma' : H \leftarrow c \wedge B \wedge R$ and $\delta' : K \leftarrow d \wedge B$ in normal form.
Output: either a clause $\gamma'' : H \leftarrow e \wedge d \wedge B \wedge R$ such that $\gamma' \cong \gamma''$ and $evars(\delta') \cap vars(e) = \emptyset$, or FAILURE.

Step 1. Transform the formula $\hat{e} : \forall Z\ (d \rightarrow c)$, where $Z = vars(c) \cap vars(d)$, into a finite disjunction of constraints as follows:

1. transform the formula into $\neg\exists Z \neg\ (d \rightarrow c)$;
2. eliminate negation from the formula $\neg\ (d \rightarrow c)$ by pushing $\neg$ inward as much as possible, and replacing $\neg(p_1 \geq p_2)$ by $p_2 > p_1$, $\neg(p_1 > p_2)$ by $p_2 \geq p_1$, and $\neg(p_1 = p_2)$ by $p_1 > p_2 \vee p_2 > p_1$ and obtain a formula $f_1 \vee \ldots \vee f_k$, where $f_1, \ldots, f_k$ are constraints;
3. distribute $\exists Z$ over $\vee$, eliminate the existential variables Z from each disjunct f_i using the function *solve*, and obtain the formula $g_1 \vee \ldots \vee g_k$, where for $i = 1, \ldots, k$ the disjunct g_i is the constraint $solve(f_i, vars(f_i) - Z)$;
4. eliminate the negation from the formula $\neg\ (g_1 \vee \ldots \vee g_k)$ by pushing $\neg$ inward, and obtain a formula $h_1 \vee \ldots \vee h_m$, where $h_1, \ldots, h_m$ are constraints;

Step 2. Let $\{e_1, \ldots, e_n\}$, with $n \leq m$, be the set of those constraints h_i such that $\mathcal{Q} \models \exists(h_i \wedge d)$. *IF* there exists e_i in $\{e_1, \ldots, e_n\}$ such that $\mathcal{Q} \models \forall(c \rightarrow \exists Y(e_i \wedge d))$, where Y is the set $fvars(\hat{e} \wedge d) - vars(\{H, B \wedge R\})$ *THEN* let e be e_i and return $\gamma'' : H \leftarrow e \wedge d \wedge B \wedge R$ *ELSE* return FAILURE.

In the following theorems, we prove that the constraint matching procedure $\mathbf{CM_1}$ is sound and terminates. Therefore, if $\mathbf{CM_1}$ constructs a constraint e then the clauses $\gamma' : H \leftarrow c \wedge B \wedge R$ and $\gamma'' : H \leftarrow e \wedge d \wedge B \wedge R$ are equivalent.

Theorem 15 (Soundness of the Constraint Matching Procedure $\mathbf{CM_1}$) *Let the two clauses $\gamma' : H \leftarrow c \wedge B \wedge R$ and $\delta' : K \leftarrow d \wedge B$ in normal form be the input of the constraint matching procedure $\mathbf{CM_1}$. If the output is the clause $\gamma'' : H \leftarrow e \wedge d \wedge B \wedge R$, then $\gamma' \cong \gamma''$, and $evars(\delta') \cap vars(e) = \emptyset$.*

Proof: By the Soundness Theorem 13, the clause γ'' is in normal form. Therefore, by Lemma 15, $H \leftarrow c \wedge B \wedge R \cong H \leftarrow e \wedge d \wedge B \wedge R$ if $\mathcal{Q} \models \forall(c \leftrightarrow \exists W\,(e \wedge d))$, where W is the set of the local variables of the constraint $e \wedge d$ in γ''. By Lemma 16 we have that $\mathcal{Q} \models \forall(\exists Y\,(\hat{e} \wedge d) \rightarrow c)$. Step 1 transforms $\hat{e}$ into a logically equivalent formula $h_1 \vee \ldots \vee h_m$. By Lemma 19 we have that $\mathcal{Q} \models \forall(\exists Y\,(e_i \wedge d) \rightarrow c)$, for $i = 1, \ldots, n$. Then, Step 2 selects an e_i such that $\mathcal{Q} \models \forall(c \rightarrow \exists Y\,(e_i \wedge d))$. So, since e is defined to be e_i, we have $\mathcal{Q} \models \forall(c \leftrightarrow \exists Y\,(e \wedge d))$. Note that, by definition of the sets W and Y, we have $W \subseteq Y$. Now, we have that $Y - W = fvars(\hat{e}) - (vars(e_i) \cup vars(\{H, B \wedge R\}))$. Hence, the variables in the set $Y - W$ do not occur in the constraint $e \wedge d$. As a consequence, we have $\mathcal{Q} \models \forall(c \leftrightarrow \exists W\,(e \wedge d))$ and, thus, $\gamma' \cong \gamma''$. Let us now consider that, by construction and by definition of $\hat{e}$, we have that $vars(e) \subseteq fvars(\hat{e}) = (vars(c) \cup vars(d)) - Z$. Also, recalling that the clause γ' is in normal form, we have that $evars(\delta') \subseteq vars(B)$. Now, suppose that $X \in vars(B)$, then $X \notin vars(d) - Z$ because the clause δ' is in normal form and $X \notin vars(c) - Z$ because the clauses γ' and δ' are the output of the Goal Matching Phase, thus $Z = vars(B)$. As a consequence, $evars(\delta') \cap vars(e) = \emptyset$. □

Theorem 16 (Termination of the Constraint Matching Procedure $\mathbf{CM_1}$) *The constraint matching procedure $\mathbf{CM_1}$ terminates for every pair of input clauses γ' and δ' in normal form.*

Proof: The operations performed in $\mathbf{CM_1}$ always terminate. □

We now introduce the folding algorithm $\mathbf{FA_1}$, which combines the goal matching procedure **GM** and the constraint matching procedure $\mathbf{CM_1}$.

In order to fold a clause γ using the clause δ, the folding algorithm $\mathbf{FA_1}$ first (i) applies the goal matching procedure **GM** and, if at all possible, computes a matching substitution ϑ and a goal R, and then (ii) it applies the constraint matching procedure $\mathbf{CM_1}$ and looks for a constraint e such that Conditions 1 and 2 of Definition 7 are satisfied. If $\mathbf{CM_1}$ fails to compute the desired constraint e, the folding algorithm executes again the goal matching procedure so that new, alternative outputs ϑ and R are generated, until the $\mathbf{CM_1}$ computes the desired constraint e. The folding algorithm $\mathbf{FA_1}$ returns FAILURE in the case where **GM** fails to compute a substitution ϑ and a goal R, and in the case where, for all ϑ and R computed by **GM**, the $\mathbf{CM_1}$ procedure fails to compute a constraint e such that Conditions 1 and 2 of Definition 7 are satisfied.

Folding Algorithm: $\mathbf{FA_1}$

Input: two clauses $\gamma\colon H \leftarrow c \wedge G$ and $\delta\colon K \leftarrow d \wedge B$ in normal form and without variables in common.

Output: either a clause $\eta\colon H \leftarrow e \wedge K\vartheta \wedge R$, such that η is the result of folding γ using δ according to Definition 7, or FAILURE.

Step 1. Execute the goal matching procedure **GM** with the clauses γ and δ as input. If the output of **GM** is FAILURE then return FAILURE else let the output of **GM** be the two clauses $\gamma'\colon H \leftarrow c \wedge B\vartheta \wedge R$ and $\delta'\colon K\vartheta \leftarrow d\vartheta \wedge B\vartheta$;

Step 2. Execute the constraint matching procedure $\mathbf{CM_1}$ with the clauses γ' and δ' as input. If the output of $\mathbf{CM_1}$ is FAILURE then go to Step 1 and search for a new pair of output clauses of **GM**;

Step 3. If no output of **GM** can be computed such that the output of $\mathbf{CM_1}$ is different from FAILURE, then return FAILURE else let the output of the $\mathbf{CM_1}$ be the clause $\gamma''\colon H \leftarrow e \wedge d\vartheta \wedge B\vartheta \wedge R$. Return $\eta\colon H \leftarrow e \wedge K\vartheta \wedge R$.

By putting together Theorems 13, 14, 15, and 16, we can prove the termination and soundness of the folding algorithm $\mathbf{FA_1}$.

Theorem 17 (Soundness of the Folding Algorithm $\mathbf{FA_1}$) *Let the two clauses γ and δ, in normal form and with no variables in common, be the input of the folding algorithm* $\mathbf{FA_1}$*. If the output is a clause η, then η can be obtained by folding γ using δ according to Definition 7.*

Proof: Let γ be $H \leftarrow c \wedge G$ and δ be $K \leftarrow d \wedge B$. We have to show that, given the constraint e, the substitution ϑ, and the goal R computed by the folding algorithm $\mathbf{FA_1}$, $H \leftarrow c \wedge G \ \cong \ H \leftarrow e \wedge d\vartheta \wedge B\vartheta \wedge R$. This is motivated by the Soundness Theorem 13, by the Soundness Theorem 15 and by Lemma 12 regarding the transitivity of the operator $\cong$. In particular, by the Soundness Theorem 13, clauses γ' and δ' generated at Step 1 are in normal form. Second, we have to show that the substitution ϑ satisfies Condition 2 of Definition 7. Note that the substitution ϑ is computed at Step 1 by the goal matching procedure **GM**. Therefore, by the Soundness Theorem 13, for every variable X in $evars(\delta)$, the following properties hold: (i) $X\vartheta$ is a variable not occurring in $\{H, R\}$, and (ii) $X\vartheta$ does not occur in the term $Y\vartheta$, for every variable Y occurring in $d \wedge B$ and different from X. Now, in order to prove that the substitution ϑ satisfies Condition 2 of Definition 7, we have to show that if X is a variable in $evars(\delta)$, then $X\vartheta$ does not occur in e. By the property (ii) mentioned above, if X is an existential variable of the clause δ, then $X\vartheta$ does not occur in the term $Y\vartheta$, for every variable Y occurring in $d \wedge B$ and different from X. As a consequence, X is an existential variable of the clause δ iff $X\vartheta$ is an existential variable of the clause δ'. Also, by the Soundness Theorem 15, $evars(\delta') \cap vars(e) = \emptyset$ and thus if X is a variable in $evars(\delta)$, then $X\vartheta$ does not occur in e. □

Theorem 18 (Termination of the Folding Algorithm $\mathbf{FA_1}$) *The Folding Algorithm* $\mathbf{FA_1}$ *terminates for every pair of input clauses in normal form and without common variables.*

Proof: The theorem follows from Theorems 14 and 16, and from the fact that there is only a finite number of different outputs of the goal matching procedure **GM**. □

Let us now illustrate the folding algorithm $\mathbf{FA_1}$ by considering the example proposed at the beginning of this chapter. We are given the following

clauses as input:

γ: $p(X_1, X_2) \leftarrow Z > 0 \wedge X_2 > Z+1 \wedge X_1 \geq 2X_2 \wedge s(X_1, Z)$
δ: $q(Y_1, Y_2, Y_3) \leftarrow Y_1+Y_3 \geq Y_2 \wedge Y_2 > 0 \wedge Y_2-2 > 2Y_3 \wedge s(Y_1 + Y_3, Y_3)$

By using the Normalization Algorithm, we rewrite the clauses γ and δ in normal form:

γ: $p(X_1, X_2) \leftarrow Z_2 > 0 \wedge X_2 > Z_2+1 \wedge X_1 \geq 2X_2 \wedge X_1 = Z_1 \wedge s(Z_1, Z_2)$
δ: $q(Y_1, Y_2, Y_3) \leftarrow Y_1+Y_3 \geq Y_2 \wedge Y_2 > 0 \wedge Y_2-2 > 2Y_3 \wedge Y_1+Y_3 = W_1 \wedge$
$Y_3 = W_2 \wedge s(W_1, W_2)$

The goal matching procedure **GM**, applied to the clauses γ and δ, returns the following pair of clauses:

γ': $p(X_1, X_2) \leftarrow Z_2 > 0 \wedge X_2 > Z_2+1 \wedge X_1 \geq 2X_2 \wedge X_1 = Z_1 \wedge s(Z_1, Z_2)$
δ': $q(Y_1, Y_2, Y_3) \leftarrow Y_1+Y_3 \geq Y_2 \wedge Y_2 > 0 \wedge Y_2-2 > 2Y_3 \wedge Y_1+Y_3 = Z_1 \wedge$
$Y_3 = Z_2 \wedge s(Z_1, Z_2)$

which are now the input of the constraint matching procedure $\mathbf{CM_1}$. Let us consider in detail the execution of $\mathbf{CM_1}$. At the end of Step 1, we obtain the following formula, which is the disjunction of four constraints, h_1, h_2, h_3, and h_4, respectively.:

$$2Y_3+2 \geq Y_2 \vee 0 \geq Y_2 \vee Y_2 > Y_3+Y_1 \vee (X_2 > 1 \wedge X_1 \geq 2X_2 \wedge X_1 \geq Y_1 \wedge Y_3+Y_1 = X_1)$$

At Step 2 we compute the formula $d|_{\{Y_1,Y_2,Y_3\}}$ and obtain the following constraint:

$$Y_1+Y_3 \geq Y_2 \wedge Y_2 > 0 \wedge Y_2-2 > 2Y_3$$

Then, by using the function *solve* we prove that $\mathcal{Q} \not\models \exists(d|_{\{Y_1,Y_2,Y_3\}} \wedge h_1)$, $\mathcal{Q} \not\models \exists(d|_{\{Y_1,Y_2,Y_3\}} \wedge h_2)$, and $\mathcal{Q} \not\models \exists(d|_{\{Y_1,Y_2,Y_3\}} \wedge h_3)$. As a consequence, at Step 2, we have the set:

$$\{e\} = \{X_2 > 1 \wedge X_1 \geq 2X_2 \wedge X_1 \geq Y_1 \wedge Y_3+Y_1 = X_1\}$$

By using the function *solve* we verify that $\mathcal{Q} \models \forall(c \rightarrow \exists Y_1 \exists Y_2 \exists Y_3(e \wedge d))$. Therefore, the output of the constraint matching procedure $\mathbf{CM_1}$ is the following clause

γ'': $p(X_1, X_2) \leftarrow X_2 > 1 \wedge X_1 \geq 2X_2 \wedge X_1 \geq Y_1 \wedge Y_3+Y_1 = X_1 \wedge Y_1+Y_3 \geq Y_2 \wedge$
$Y_2 > 0 \wedge Y_2-2 > 2Y_3 \wedge Y_1+Y_3 = Z_1 \wedge Y_3 = Z_2 \wedge s(Z_1, Z_2)$

Finally, the output of the Folding Algorithm is the clause:

$$\eta\colon\ p(X_1, X_2) \leftarrow X_2 > 1 \wedge X_1 \geq 2X_2 \wedge X_1 \geq Y_1 \wedge Y_3 + Y_1 = X_1 \wedge q(Y_1, Y_2, Y_3)$$

Due to the the fact that, in general, the constraint matching procedure $\mathbf{CM_1}$ is not complete, the folding algorithm $\mathbf{FA_1}$ is not complete either. Indeed, for some pairs of input clauses γ and δ there exists a suitable triple $\langle e, \vartheta, R\rangle$ such that Conditions (1) and (2) of Definition 7 are satisfied, and yet, $\mathbf{FA_1}$ returns FAILURE.

In the next two sections, we will present two complete folding algorithms for applying the standard folding rule as introduced in Definition 7, if some conditions arise. In particular, the first algorithm can be used when preliminary applications of the clause splitting rule are possible, while the second algorithm can be used when there is the extra requirement that no existential variable should be present in the folded rule.

4.3 A Folding Rule Combined with Clause Splitting

In this section we present a rule, called $\mathbf{Fold_2}$, which is an extension of the folding rule $\mathbf{Fold_1}$ presented in Section 4.2. An application of the rule $\mathbf{Fold_2}$ is equivalent to zero or more applications of the clause splitting rule [27] followed by several applications of the folding rule $\mathbf{Fold_1}$.

Definition 9 below introduces the folding rule $\mathbf{Fold_2}$ by which a clause $\gamma\colon H \leftarrow c \wedge G$ can be folded using a clause $\delta\colon K \leftarrow d \wedge B$ if: (i) there exist n constraints $c_1, \ldots, c_n$ such that $\mathcal{Q} \models \forall(c \leftrightarrow c_1 \vee \ldots \vee c_n)$ holds, and (ii) for $i = 1, \ldots, n$, by applying the folding rule $\mathbf{Fold_1}$, the clause $H \leftarrow c_i \wedge G$ can be folded using δ, thereby deriving $H \leftarrow e_i \wedge K\vartheta \wedge R$.

In the definition of the folding rule $\mathbf{Fold_2}$, we will not separate the application of the clause splitting rule from the applications of the folding rule. In particular, we will not require to find the intermediate constraints $c_1, \ldots, c_n$ in an explicit way, but we will only require to find the final constraints $e_1, \ldots, e_n$, the substitution ϑ, and the goal R.

Definition 9 (Rule $\mathbf{Fold_2}$)
Let γ and δ be clauses of the form

γ: $H \leftarrow c \wedge G$
δ: $K \leftarrow d \wedge B$

such that γ and δ are in normal form and have no common variables. Suppose that there exist some constraints $e_1, \ldots, e_n$, a substitution ϑ, and a goal R such that:
1) $\{H \leftarrow c \wedge G\} \cong \{H \leftarrow e_1 \wedge d\vartheta \wedge B\vartheta \wedge R, \ldots, H \leftarrow e_n \wedge d\vartheta \wedge B\vartheta \wedge R\}$;
2) for every variable X in $evars(\delta)$, the following conditions hold: (i) $X\vartheta$ is a variable not occurring in $\{H, e_i, R\}$, for $i = 1, \ldots, n$, and (ii) $X\vartheta$ does not occur in the term $Y\vartheta$, for every variable Y occurring in $d \wedge B$ and different from X.
By *folding clause* γ *using clause* δ we derive the clauses $\eta_1 : H \leftarrow e_1 \wedge K\vartheta \wedge R, \ldots, \eta_n : H \leftarrow e_n \wedge K\vartheta \wedge R$.

Now we provide an algorithm, called $\mathbf{FA_2}$, for applying the $\mathbf{Fold_2}$ rule. Given two input clauses γ and δ, the output of the algorithm is a set of clauses derived by folding γ using δ as specified in Definition 9, if it is possible to fold, and the output is FAILURE, otherwise. The $\mathbf{FA_2}$ algorithm looks for suitable constraints $e_1, \ldots, e_n$, substitution ϑ, and goal R such that Conditions (1) and (2) of Definition 9 hold.

Similarly to Algorithm $\mathbf{FA_1}$ presented in Section 4.2, Algorithm $\mathbf{FA_2}$ makes use of two procedures: the goal matching procedure **GM**, which is the one also used in $\mathbf{FA_1}$, and the constraint matching procedure $\mathbf{CM_2}$, which is the following variant of $\mathbf{CM_1}$.

Constraint Matching Phase: $\mathbf{CM_2}$

Input: two clauses $\gamma' : H \leftarrow c \wedge B \wedge R$ and $\delta' : K \leftarrow d \wedge B$ in normal form.

Output: (a) a set $\{\gamma''_1, \ldots, \gamma''_n\}$ of clauses, with $n \geq 0$, such that: (i) for $i = 1, \ldots, n$, γ''_i is of the form $H \leftarrow e_i \wedge d \wedge B \wedge R$, (ii) $\{\gamma'\} \cong \{\gamma''_1, \ldots, \gamma''_n\}$, and (iii) $evars(\delta') \cap vars(\{e_1, \ldots, e_n\}) = \emptyset$, if such set exists, and (b) FAILURE, otherwise.

Step 1. Compute the disjunction $h_1 \vee \ldots \vee h_m$ of constraints in the same way as at Step 1 of the $\mathbf{CM_1}$ procedure;

Step 2. Let $\{e_1, \ldots, e_n\}$, with $n \leq m$, be the set of those h_i such that, for $i = 1, \ldots, m$, $\mathcal{Q} \models \exists(h_i \wedge d)$. *IF* $\mathcal{Q} \models \forall(c \rightarrow \exists Y\,((e_1 \vee \ldots \vee e_n) \wedge d))$, where $Y = \mathit{fvars}(\hat{e} \wedge d) - \mathit{vars}(\{H, B \wedge R\})$ *THEN* return the set $\{H \leftarrow e_i \wedge d \wedge B \wedge R \mid i = 1, \ldots, n\}$ of clauses ELSE return FAILURE.

In the following theorems, we prove that the constraint matching procedure $\mathbf{CM_2}$ is sound and complete with respect to its Input/Output specification.

Theorem 19 (Soundness of the Constraint Matching Procedure $\mathbf{CM_2}$) *Let the two clauses $\gamma' : H \leftarrow c \wedge B \wedge R$ and $\delta' : K \leftarrow d \wedge B$ in normal form be the input of the constraint matching procedure $\mathbf{CM_2}$. If the output is the set $\{\gamma''_1, \ldots, \gamma''_n\}$ of clauses where, for $i = 1, \ldots, n$, $\gamma''_i : H \leftarrow e_i \wedge d \wedge B \wedge R$, then $\{\gamma'\} \cong \{\gamma''_1, \ldots, \gamma''_n\}$ and $\mathit{evars}(\delta') \cap \mathit{vars}(\{e_1, \ldots, e_n\}) = \emptyset$.*

Proof: We want to show that $\{\gamma'\} \cong \{\gamma''_1, \ldots, \gamma''_n\}$. By Lemma 15 we have $\{H \leftarrow c \wedge B \wedge R\} \cong \{H \leftarrow e_1 \wedge d \wedge B \wedge R, \ldots, H \leftarrow e_n \wedge d \wedge B \wedge R\}$ iff $\mathcal{Q} \models \forall(c \leftrightarrow (\exists W_1\,(e_1 \wedge d) \vee \ldots \vee \exists W_n\,(e_n \wedge d)))$, where W_i is the set of local variables of $e_i \wedge d$ in γ''_i, for $i = 1, \ldots, n$. The constraint matching procedure $\mathbf{CM_2}$ starts from the formula $\hat{e}$ and, at Step 1, rewrites it as the logically equivalent formula $h_1 \vee \ldots \vee h_m$. Then, at Step 2, performs some simplifications transforming the formula $h_1 \vee \ldots \vee h_m$ into the logically equivalent formula $e_1 \vee \ldots \vee e_n$ and tests whether $\mathcal{Q} \models \exists(c \rightarrow \exists Y\,((e_1 \vee \ldots \vee e_n) \wedge d))$, where $Y = \mathit{fvars}(\hat{e} \wedge d) - \mathit{vars}(\{H, B \wedge R\})$, or not. In the former case, by Lemma 18, we have that $\mathcal{Q} \models \forall\,(c \leftrightarrow \exists Y((e_1 \vee \ldots \vee e_n) \wedge d))$. Therefore, by distributing the existential quantification over $\vee$ we obtain $\mathcal{Q} \models \forall\,(c \leftrightarrow (\exists Y(e_1 \wedge d) \vee \ldots \vee \exists Y(e_n \wedge d)))$. Note that, by definition, $W_i \subseteq Y$ and $Y - W_i = \mathit{fvars}(\hat{e}) - (\mathit{vars}(e_i) \cup \mathit{vars}(\{H, B \wedge R\}))$. Thus, the variables in the set $Y - W_i$ do not occur in the constraint $e_i \wedge d$, for $i = 1, \ldots, n$. As a consequence, we have that the equivalence $\{\gamma'\} \cong \{\gamma''_1, \ldots, \gamma''_n\}$ holds. Finally, reasoning as in the proof of the Soundness Theorem 15, we can show that $\mathit{evars}(\delta') \cap \mathit{vars}(e_i) = \emptyset$, for $i = 1, \ldots, n$. □

Theorem 20 (Completeness of the Constraint Matching Procedure $\mathbf{CM_2}$) *The constraint matching procedure* $\mathbf{CM_2}$ *terminates for all input clauses. Moreover, if the input clauses are* $\gamma' : H \leftarrow c \wedge B \wedge R$ *and* $\delta' : K \leftarrow d \wedge B$ *and it is possible to find some constraints* $e_1, \ldots, e_n$ *such that* $\{H \leftarrow c \wedge B \wedge R\} \cong \{H \leftarrow e_1 \wedge d \wedge B \wedge R, \ldots, H \leftarrow e_n \wedge d \wedge B \wedge R\}$ *and* $evars(\delta') \cap vars(\{e_1, \ldots, e_n\}) = \emptyset$*, then the output of* $\mathbf{CM_2}$ *is not FAILURE.*

Proof: The termination of the constraint matching procedure $\mathbf{CM_2}$ is straightforward, because the transformations performed at Step 1, including the application of the *solve* function, and the satisfiability tests performed at Step 2 always terminate. Let us now assume that, for some input clauses γ' and δ', there exist some constraints $e_1, \ldots, e_n$ such that $\{H \leftarrow c \wedge B \wedge R\} \cong \{H \leftarrow e_1 \wedge d \wedge B \wedge R, \ldots, H \leftarrow e_n \wedge d \wedge B \wedge R\}$ and $evars(\delta') \cap vars(\{e_1, \ldots, e_n\}) = \emptyset$. Suppose, by contradiction, that $\mathbf{CM_2}$ returns FAILURE. By Lemma 15 we have that $\mathcal{Q} \models \forall(c \rightarrow (\exists W_1 (e_1 \wedge d) \vee \ldots \vee \exists W_n (e_n \wedge d)))$. By noticing that $(W_1 \cup \ldots \cup W_n) \subseteq Y$, we have also $\mathcal{Q} \models \forall(c \rightarrow \exists Y ((e_1 \vee \ldots \vee e_n) \wedge d))$. Also, by Lemma 17, we have that $\mathcal{Q} \models \forall((e_1 \vee \ldots \vee e_n) \rightarrow \hat{e})$. Hence, $\mathcal{Q} \models \forall(c \rightarrow (\exists Y (\hat{e} \wedge d))$, which contradicts the hypothesis. Therefore, the $\mathbf{CM_2}$ is complete. □

Now, we slightly modify the folding algorithm $\mathbf{FA_1}$ by using $\mathbf{CM_2}$, instead of $\mathbf{CM_1}$, and we get a terminating, sound, complete folding algorithm $\mathbf{FA_2}$ to compute the result of applying Rule $\mathbf{Fold_2}$ to clause γ using clause δ.

Folding Algorithm: $\mathbf{FA_2}$

Input: two clauses $\gamma : H \leftarrow c \wedge G$ and $\delta : K \leftarrow d \wedge B$ in normal form and with no common variables;

Output: the clauses $\eta_i : H \leftarrow e_i \wedge K\vartheta \wedge R$, for $i = 1, \ldots, n$, if it is possible to fold γ using δ according to Definition 9, and FAILURE, otherwise.

Step 1. Execute the goal matching procedure **GM** presented in Section 4.2 with the clauses γ and δ as input. If the output of **GM** is

FAILURE then return FAILURE else let the output of **GM** be the two clauses $\gamma' : H \leftarrow c \wedge B\vartheta \wedge R$ and $\delta' : K\vartheta \leftarrow d\vartheta \wedge B\vartheta$;

Step 2. Execute the constraint matching procedure $\mathbf{CM_2}$ with the clauses γ' and δ' as input. If the output is FAILURE then go to Step 1 and search for a new pair of output clauses of **GM**;

Step 3. If no output of **GM** can be found such that the output of $\mathbf{CM_2}$ is different from FAILURE, then return FAILURE else let the output of $\mathbf{CM_2}$ be the clauses $\gamma''_i : H \leftarrow e_i \wedge d\vartheta \wedge B\vartheta \wedge R$, for $i = 1, \ldots, n$. Return the clauses $\eta_i : H \leftarrow e_i \wedge K\vartheta \wedge R$, for $i = 1, \ldots, n$.

Note that, when the constraint c in the input clause γ is unsatisfiable, the output of the folding algorithm $\mathbf{FA_2}$ consists of an empty set of clauses (that is, $n = 0$). In this case (which is distinct from the case where $\mathbf{FA_2}$ returns FAILURE) an application of Rule $\mathbf{Fold_2}$ amounts to the deletion of clause γ.

In the following theorems, we show that the Folding Algorithm $\mathbf{FA_2}$ is sound and complete w.r.t. the $\mathbf{Fold_2}$ rule introduced in Definition 9.

Theorem 21 (Soundness of the Folding Algorithm $\mathbf{FA_2}$) *Let the two clauses $\gamma : H \leftarrow c \wedge G$ and $\delta : K \leftarrow d \wedge B$, in normal form and without common variables, be the input of the Folding Algorithm $\mathbf{FA_2}$. If the Folding Algorithm $\mathbf{FA_2}$ returns the clauses $\eta_i : H \leftarrow e_i \wedge K\vartheta \wedge R$, for $i = 1, \ldots, n$, then the clauses $\eta_1, \ldots, \eta_n$ can be obtained by folding γ using δ according to Definition 9.*

Proof: Let γ and δ be the input clauses of the Folding Algorithm $\mathbf{FA_2}$ and $\eta_1, \ldots, \eta_n$ be the output clauses. In order to satisfy Definition 9, we have to show that $\{H \leftarrow c \wedge G\} \cong \{H \leftarrow e_1 \wedge d\vartheta \wedge B\vartheta \wedge R, \ldots, H \leftarrow e_n \wedge d\vartheta \wedge B\vartheta \wedge R\}$, which holds by the Soundness Theorems 13 and 19, and by Lemma 12, regarding the transitivity of the operator $\cong$. In particular, Theorem 19 holds because, by the Soundness Theorem 13, the clauses γ' and δ' are in normal form. Also, we have to show that the substitution ϑ satisfies Condition 2 of Definition 9. By using the same observations

provided in the proof of the Soundness Theorem 17, we can show that the substitution ϑ computed in the goal matching procedure **GM** is such that if X is a variable in $evars(\delta)$, then $X\vartheta$ does not occur in e_i, for $i = 1, \ldots, n$. As a consequence, the clauses $\eta_1, \ldots, \eta_n$ can be obtained by folding γ using δ according to Definition 9. □

Theorem 22 (Completeness of the Folding Algorithm $\mathbf{FA_2}$) *Let the two clauses $\gamma : H \leftarrow c \wedge G$ and $\delta : K \leftarrow d \wedge B$, in normal form and without common variables, be the input of the Folding Algorithm $\mathbf{FA_2}$, then the algorithm terminates. Moreover, if it is possible to fold γ using δ according to Definition 9, then the output of the Folding Algorithm $\mathbf{FA_2}$ is the clauses $\eta_1, \ldots, \eta_n$, that is the output is not FAILURE.*

Proof: By the assumption that γ and δ are in normal form and without common variables, and by the Completeness Theorems 14 and 20, each execution of the goal matching procedure **GM** and of the constraint matching procedure $\mathbf{CM_2}$ terminates. By using the arguments proposed in the proof of Theorem 14, we can show that the number of possible mappings μ computed during **GM** and, therefore, the number of different output clauses is finite. Thus, the Folding Algorithm $\mathbf{FA_2}$ terminates.

Assume, now, that there exist some constraints $e'_1, \ldots, e'_n$, a goal R' and a substitution ϑ' such that $\{H \leftarrow c \wedge G\} \cong \{H \leftarrow e'_1 \wedge d\vartheta' \wedge B\vartheta' \wedge R', \ldots, H \leftarrow e'_n \wedge d\vartheta' \wedge B\vartheta' \wedge R'\}$, and ϑ' satisfies Condition 2 of Definition 9. Assume also that, given the clauses γ and δ as input, the Folding Algorithm $\mathbf{FA_2}$ returns FAILURE. Then, either **GM** fails to compute a substitution ϑ and a goal R or, for all possible output clauses computed by **GM**, $\mathbf{CM_2}$ fails to compute some constraints $e_1, \ldots, e_n$. Let us consider the first case. By the Completeness Theorem 14, **GM** computes all possible ϑ and R such that $G =_{AC} B\vartheta \wedge R$. Therefore, by the assumptions, the output of **GM** is not FAILURE. In the second case, by hypothesis we have that there exist some some constraints $e'_1, \ldots, e'_n$ such that $evars(\delta) \cap vars(\{e'_1, \ldots, e'_n\}) = \emptyset$ and $\{\gamma\} \cong \{\gamma''_1, \ldots, \gamma''_n\}$, and the output of $\mathbf{CM_2}$ is FAILURE. However, by the Soundness Theorem 13 we have that γ' and δ' are in normal form and by the transitivity of $\cong$ we have that $\{\gamma'\} \cong \{\gamma''_1, \ldots, \gamma''_n\}$. Also recalling that, by

the Soundness Theorem 13, the substitution ϑ is such that $X \in \mathit{evars}(\delta)$ iff $X \in \mathit{evars}(\delta\vartheta)$, that is $X \in \mathit{evars}(\delta)$ iff $X \in \mathit{evars}(\delta')$, we have that $\mathit{evars}(\delta') \cap \mathit{vars}(\{e'_1, \ldots, e'_n\}) = \emptyset$. Under these conditions, the Completeness Theorem 20 states that the output of $\mathbf{CM_2}$ is different from FAILURE. As a consequence, the Folding Algorithm $\mathbf{FA_2}$ is complete. □

Let us now illustrate the folding algorithm $\mathbf{FA_2}$ by considering again the second example proposed in the beginning of this chapter. We are given the following clauses as input:

γ: $p(X) \leftarrow 2 \geq X \wedge X \geq 1 \wedge Z \geq 1 \wedge 2 \geq Z \wedge q(Z)$
δ: $s(Y) \leftarrow W + \frac{3}{4} \geq Y \wedge Y \geq W - \frac{3}{4} \wedge 4 - W \geq Y \wedge Y \geq 2 - W \wedge q(W)$

These clauses are already in normal form. The goal matching procedure **GM**, applied to the clauses γ and δ, returns the following pair of clauses:

γ': $p(X) \leftarrow 2 \geq X \wedge X \geq 1 \wedge Z \geq 1 \wedge 2 \geq Z \wedge q(Z)$
δ': $s(Y) \leftarrow Z + \frac{3}{4} \geq Y \wedge Y \geq Z - \frac{3}{4} \wedge 4 - Z \geq Y \wedge Y \geq 2 - Z \wedge q(Z)$

Let us now consider in detail the execution of the constraint matching procedure $\mathbf{CM_2}$ applied to the clauses γ' and δ'. At the end of Step 1, we obtain the following formula, which is the disjunction of four constraints, h_1, h_2, h_3, and h_4, respectively. :

$$Y > \tfrac{19}{8} \vee Y < \tfrac{5}{8} \vee (2 \geq X \wedge X \geq 1 \wedge Y \geq 2) \vee (2 \geq X \wedge X \geq 1 \wedge 1 \geq Y)$$

At Step 2 we compute the formula $d|_{\{Y_1, Y_2, Y_3\}}$ and obtain the following constraint:

$$\tfrac{19}{8} \geq Y \vee Y \geq \tfrac{5}{8}$$

Further, by using the function *solve* we prove that $\mathcal{Q} \not\models \exists(d|_{\{Y_1, Y_2, Y_3\}} \wedge h_1)$, $\mathcal{Q} \not\models \exists(d|_{\{Y_1, Y_2, Y_3\}} \wedge h_2)$. As a consequence, at Step 2, we have the set:

$$\{e_1, e_2\} = \{2 \geq X \wedge X \geq 1 \wedge Y \geq 2, 2 \geq X \wedge X \geq 1 \wedge 1 \geq Y\}$$

By using the function *solve* we verify that $\mathcal{Q} \models \forall(c \rightarrow \exists Y(e_1 \wedge d))$ and $\mathcal{Q} \models \forall(c \rightarrow \exists Y(e_2 \wedge d))$. Therefore, the output of the constraint matching procedure $\mathbf{CM_2}$ is the set $\{\gamma''_1, \gamma''_2\}$ of clauses, defined as follows:

γ_1'': $p(X) \leftarrow 2 \geq X \wedge X \geq 1 \wedge Y \geq 2 \wedge Z + \frac{3}{4} \geq Y \wedge Y \geq Z - \frac{3}{4} \wedge$
$4 - Z \geq Y \wedge Y \geq 2 - Z \wedge q(Z)$
γ_1'': $p(X) \leftarrow 2 \geq X \wedge X \geq 1 \wedge 1 \geq Y \wedge Z + \frac{3}{4} \geq Y \wedge Y \geq Z - \frac{3}{4} \wedge$
$4 - Z \geq Y \wedge Y \geq 2 - Z \wedge q(Z)$

Finally, the output of the folding algorithm $\mathbf{FA_2}$ is the pair of clauses:

η_1: $p(X) \leftarrow 2 \geq X \wedge X \geq 1 \wedge Y \geq 2 \wedge s(Y)$
η_2: $p(X) \leftarrow 2 \geq X \wedge X \geq 1 \wedge 1 \geq Y \wedge s(Y)$

4.4 A Folding Rule for the Elimination of Existential Variables

In the field of logic programs transformation the elimination of existential variables is a technique of interest for several applications. In particular, it can be used to achieve program optimizations and to develop techniques for automated theorem proving. Indeed, in this field the well known technique of quantifier elimination can be applied by properly encoding a logic formula as a logic program, and then eliminating the existential variables of this program. In Chapter 3 we illustrate an application of this theorem proving technique in detail.

In this section we introduce a folding rule, called $\mathbf{Fold_3}$, which is a variant of the $\mathbf{Fold_1}$ rule tailored to the elimination of existential variables. When we apply the $\mathbf{Fold_3}$ rule to a pair of input clauses $\gamma : H \leftarrow c \wedge G$ and $\delta : K \leftarrow d \wedge B$, we replace a subgoal of $c \wedge G$, where some existential variables may occur, by an atom $K\vartheta$ without existential variables.

Definition 10 (Rule $\mathbf{Fold_3}$)
Let γ and δ be clauses of the form

γ: $H \leftarrow c \wedge G$
δ: $K \leftarrow d \wedge B$

such that γ and δ have no common variables. Suppose that there exists a constraint e, a substitution ϑ, and goal R such that:
1) $H \leftarrow c \wedge G \ \cong \ H \leftarrow e \wedge d\vartheta \wedge B\vartheta \wedge R$;

2) for every variable X in $evars(\delta)$, the following conditions hold: (i) $X\vartheta$ is a variable not occurring in $\{H, e, R\}$, and (ii) $X\vartheta$ does not occur in the term $Y\vartheta$, for every variable Y occurring in $d \wedge B$ and different from X;
3) $vars(K\vartheta) \subseteq vars(H)$.
By *folding clause* γ *using clause* δ we derive the clause η: $H \leftarrow e \wedge K\vartheta \wedge R$.

Condition (3) ensures that all the existential variables occurring in the folded subgoal $d\vartheta \wedge B\vartheta$ of body of the clause γ are indeed eliminated by folding. However, in the folded clause η there can be still some existential variables occurring in e and R, which can be eliminated by further folding steps as the following example shows. Suppose that we are given the following clauses γ_1 and δ_1:

γ_1: $p(X) \leftarrow X = Z_1 \wedge X = Z_3 \wedge X > Z_2 \wedge X > Z_4 \wedge q(Z_1, Z_2) \wedge r(Z_3, Z_4)$
δ_1: $s(Y_1) \leftarrow Y_1 = W_1 \wedge Y_1 > W_2 \wedge q(W_1, W_2)$

A possible folding of γ_1 using δ_1 is determined by: (i) the constraint e_1: $X = Z_3 \wedge X > Z_4$, (ii) the substitution ϑ_1: $\{Y_1/X,\ W_1/Z_1,\ W_2/Z_2\}$, and (iii) the goal R_1: $r(Z_3, Z_4)$. The clause derived by folding γ_1 using δ_1 is:

η_1: $p(X) \leftarrow X = Z_3 \wedge X > Z_4 \wedge s(X) \wedge r(Z_3, Z_4)$

Note that, $vars(s(X)) \subseteq vars(p(X))$, as required by Condition (3) of Definition 10. However, in clause η_1 there are still some occurrences of existential variables, namely Z_3 and Z_4. Let the following clause γ_2 be a normal form of the clause η_1:

γ_2: $p(X) \leftarrow X = Z_3 \wedge X > Z_4 \wedge X = Z_5 \wedge s(Z_5) \wedge r(Z_3, Z_4)$

and let the following clause δ_2 be in normal form and such that γ_2 and δ_2 have no common variables:

δ_2: $s(Y_2) \leftarrow Y_2 = W_3 \wedge Y_2 > W_4 \wedge q(W_3, W_4)$

A possible folding of γ_2 using δ_2 is determined by: (i) the constraint e_2: $X = Z_5$, (ii) the substitution ϑ_2: $\{Y_2/X,\ W_3/Z_3,\ W_4/Z_4\}$, and (iii) the goal R_2: $r(Z_3, Z_4)$. The clause derived by folding γ_2 using δ_2 is:

η_2: $p(X) \leftarrow X = Z_5 \wedge s(Z_5) \wedge t(X)$

which, by eliminating the equality $X = Z_5$, can be transformed into the following clause without existential variables:

η_3: $p(X) \leftarrow s(X) \wedge t(X)$

In the rest of this section we present an algorithm, called $\mathbf{FA_3}$, for applying Rule $\mathbf{Fold_3}$. The $\mathbf{FA_3}$ algorithm is a variant of the $\mathbf{FA_1}$ algorithm for applying Rule $\mathbf{Fold_1}$ presented in Section 4.2. Given two clauses γ and δ, the $\mathbf{FA_3}$ algorithm returns a clause obtained by folding γ using δ according to Definition 10, if it is possible to fold, and it returns FAILURE, otherwise. The $\mathbf{FA_3}$ algorithm makes use of the goal matching procedure **GM** presented in Section 4.2 and of a constraint matching procedure, called $\mathbf{CM_3}$. The $\mathbf{CM_3}$ procedure takes as input the two clauses $\gamma'\colon H \leftarrow c \wedge B \wedge R$ and $\delta'\colon K \leftarrow d \wedge B$, which are the output of the goal matching procedure. If it is possible to find a constraint e and a substitution ϑ_2 such that (i) $B\vartheta_2 = B$, (ii) $vars(K\vartheta_2) \subseteq vars(H)$, (iii) $vars(e) \subseteq vars(H, R)$ and (iv) $H \leftarrow c \wedge B \wedge R \cong H \leftarrow e \wedge d\vartheta_2 \wedge B \wedge R$, then the constraint matching procedure $\mathbf{CM_3}$ returns a clause $\gamma''\colon H \leftarrow e \wedge d\vartheta_2 \wedge B\vartheta_2 \wedge R$, else it returns FAILURE.

First we show (see Lemma 20) that if there exists a constraint e satisfying Points (iii) and (iv) above, then we can always take such constraint to be $solve(c, X)$, where $X = vars(c) - vars(B)$.

Definition 11 Let the constraint $\tilde{e}$ be defined as $c|_X$, where c is the constraint occurring in the input clause $\gamma'\colon H \leftarrow c \wedge B \wedge R$ of the constraint matching procedure $\mathbf{CM_3}$, and $X = vars(c) - vars(B)$.

By Lemma 15, the equivalence $H \leftarrow c \wedge B \wedge R \cong H \leftarrow e \wedge d\vartheta_2 \wedge B \wedge R$ holds iff $\mathcal{Q} \models \forall(c \leftrightarrow \exists W\,(e \wedge d\vartheta_2))$, where $W = vars(e \wedge d\vartheta_2) - vars(\{H, B\vartheta_2 \wedge R\})$. In particular, in the proof of the Soundness Theorem 24 we will show that $\mathcal{Q} \models \forall(c \leftrightarrow \exists W\,(e \wedge d\vartheta_2))$ iff $\mathcal{Q} \models \forall(c \leftrightarrow e \wedge d\vartheta_2)$. Now, in the following Lemma 20, we show that if we take the constraint e to be $\tilde{e}$ then we preserve the completeness of the folding algorithm.

Lemma 20 *Let $\gamma'\colon H \leftarrow c \wedge B \wedge R$ and $\delta'\colon K \leftarrow d \wedge B$ be the input clauses of the constraint matching procedure $\mathbf{CM_3}$. If there exist a constraint e and a substitution ϑ_2 such that $\mathcal{Q} \models \forall(c \leftrightarrow (e \wedge d\vartheta_2))$, then $\mathcal{Q} \models \forall(c \leftrightarrow (\tilde{e} \wedge d\vartheta_2))$.*

Proof: Let us assume there exist a constraint e and a substitution ϑ_2 such that $\mathcal{Q} \models \forall(c \leftrightarrow e \wedge d\vartheta_2)$. By applying some logical equivalences we obtain $\mathcal{Q} \models \forall(c \rightarrow e) \wedge \forall(c \rightarrow d\vartheta_2) \wedge \forall(e \wedge d\vartheta_2 \rightarrow c)$. Then, by hypothesis, $vars(e) \subseteq vars(H, R)$ and so this implies $\mathcal{Q} \models \forall(\exists Z\, c \rightarrow e)$, where $Z = vars_c(B)$. As a consequence, we have that $\mathcal{Q} \models \forall(\exists Z\, c \wedge d\vartheta_2 \rightarrow c)$. On the other hand, $\mathcal{Q} \models \forall(c \rightarrow d\vartheta_2)$ implies that $\mathcal{Q} \models \forall(c \rightarrow \exists Z\, c \wedge d\vartheta_2)$. Thus, recalling that by definition of the function *solve* the constraint $\tilde{e}$ is equivalent to the formula $\exists Z\, c$, we obtain that $\mathcal{Q} \models \forall(c \leftrightarrow (\tilde{e} \wedge d\vartheta_2))$. □

As a consequence of these observation and in particular of Lemma 20, the constraint matching procedure **CM$_3$** reduces to the search for a substitution ϑ_2 such that $\mathcal{Q} \models \forall(c \leftrightarrow (\tilde{e} \wedge d\vartheta_2))$.

Now we introduce some notions and we establish some properties (see Lemma 21 and Theorem 23 below) which will be exploited by the constraint matching procedure **CM$_3$**.

We say that a conjunction $a_1 \wedge \ldots \wedge a_m$ of atomic constraints is *non-redundant* iff there is no constraint $a_i \in \{a_1, \ldots, a_n\}$ such that $\mathcal{Q} \models \forall(a_1 \wedge \ldots \wedge a_{i-1} \wedge a_{i+1} \wedge \ldots \wedge a_m \rightarrow a_i)$. In the following Lemma 21, we show that two given non-redundant constraints a and b are equivalent iff there exists a bijective mapping μ from the atomic constraints in a to the atomic constraints in b such that, for every atomic constraint a' occurring in a, holds that $\mathcal{Q} \models \forall\,(a' \leftrightarrow \mu(a'))$.

Lemma 21 *Let a and b be two non-redundant constraints of the form $a_1 \wedge \ldots \wedge a_m$ and $b_1 \wedge \ldots \wedge b_n$, respectively, where for $i = 1, \ldots, m$ and $j = 1, \ldots, n$, a_i and b_j are atomic constraints. Then $\mathcal{Q} \models \forall\,(a \leftrightarrow b)$ iff there exists a bijective mapping μ from the atomic constraints in a to the atomic constraints in b such that, for $i = 1, \ldots, m$, $\mathcal{Q} \models \forall\,(a_i \leftrightarrow \mu(a_i))$.*

Proof: Let us assume that $a_1 \wedge \ldots \wedge a_m$ and $b_1 \wedge \ldots \wedge b_n$ are non-redundant constraints. By simple rewritings, from the formula $a_1 \wedge \ldots \wedge a_m \leftrightarrow b_1 \wedge \ldots \wedge b_n$ we obtain a disjunction $f_1 \vee \ldots \vee f_h$, where, for $k = 1, \ldots, h$, f_k is a finite conjunction of implications of the form $a_i \rightarrow b_j$ or $b_j \rightarrow a_i$, for some atomic constraints $a_i \in \{a_1, \ldots, a_m\}$ and $b_j \in \{b_1, \ldots, b_n\}$. By construction, for $k = 1, \ldots, h$ and for $i = 1, \ldots, m$, a_i occurs in f_k two

times, one as the premise of an implication and another as the conclusion. So, let the two implications $a_i \rightarrow b_q$ and $b_p \rightarrow a_i$ occur in f_k. If p is different from q, then by transitivity we can deduce, the implication $b_p \rightarrow b_q$, which by hypothesis is equivalent to *false*, as the constraints a and b are non-redundant. Otherwise, if p is equal to q, the two implications are $a_i \rightarrow b_p$ and $b_p \rightarrow a_i$, and there is no other occurrence of a_i in f_k. The same holds symmetrically for the occurrences of b_j for $j = 1, \ldots, n$ in f_k. As a consequence, the disjunction $f_1 \vee \ldots \vee f_h$ is such that, for $k = 1, \ldots, h$ either for all $i = 1, \ldots, m$ the only occurrences of a_i in f_k are in two implications of the form $a_i \rightarrow b_j$ and $b_j \rightarrow a_i$ for some $b_j \in \{b_1, \ldots, b_n\}$, or f_k is equivalent to *false*. The same holds symmetrically for the occurrences of d_j, for $j = 1, \ldots, n$, in f_k. Thus, we obtain the formula $g_1 \vee \ldots \vee g_l$, where for $k = 1, \ldots, l$ the disjunct g_k is a finite conjunction of equivalences of the form $a_i \leftrightarrow b_j$, where a_i and b_j are atomic constraints in a and b, respectively. Also, g_k is such that, for all $i = 1, \ldots, m$, there exists only one equivalence $a_i \leftrightarrow b_j$ in g_k containing the atomic constraint a_i and b_j, and these constraints do not appear anywhere else in g_k. This is equivalent to state that, for a given $k = 1, .., l$, the set $\{(a_i, b_j) \mid a_i \leftrightarrow b_j \text{ is in } g_k\}$ is a bijection from the set $\{a_1, \ldots, a_m\}$ to the set $\{b_1, \ldots, b_n\}$. □

In the constraint matching procedure **CM$_3$** we will use Lemma 21 for finding, if it exists, a substitution ϑ_2 such that $\mathcal{Q} \models \forall(c \leftrightarrow (\tilde{e} \wedge d\vartheta_2))$, where $\tilde{e}$ is defined as in Definition 11. Indeed, given the clauses $\gamma' : H \leftarrow c \wedge B \wedge R$ and $\delta' : K \leftarrow d \wedge B$ in normal form, the **CM$_3$** procedure: (1) first computes two non-redundant constraints c' and d' equivalent to c and $\tilde{e} \wedge d$, respectively, (2) then computes an injective mapping μ from the set of the atomic constraints occurring in c' to the set of the atomic constraints occurring in d', and (3) finally, computes a substitution ϑ_2 such that: (3.i) for every atomic constraint a occurring in c', $\mathcal{Q} \models \forall(a \leftrightarrow \mu(a)\vartheta_2)$, and (3.ii) for every atomic constraint b occurring in d' and not in $\mu(c')$, where $\mu(c')$ is the conjunction of all the atomic constraints in the image of μ, we have that $\mathcal{Q} \models \forall(c' \rightarrow b\vartheta_2)$. Thus, μ is a bijective mapping from the set of the atomic constraints occurring in c' to the set of the atomic constraints occurring in $\mu(c')$ and we have $\mathcal{Q} \models \forall(c' \leftrightarrow d'\vartheta_2)$.

With regard to Point (3.i), in order to compute a substitution ϑ_2 such that, for an atomic constraint a, $\mathcal{Q} \models \forall(a \leftrightarrow \mu(a)\vartheta_2)$, we make use of the following simple property: $\mathcal{Q} \models \forall(t\,R\,0 \leftrightarrow u\,R\,0)$, where t and u are linear polynomials and $R \in \{\geq, >\}$, iff there exists $k > 0$ such that $kt - u = 0$.

With regard to Point (3.ii), in order to compute a substitution ϑ_2 such that $\mathcal{Q} \models \forall(c' \rightarrow b\vartheta_2)$, we make use of a generalization of the above mentioned simple property. This more general property of constraints is stated and proved in the following Lemma 22 and Theorem 23.

Lemma 22 *Let $k_1, \ldots, k_{n+1} \in \mathbb{Q}$ and suppose that for all $x_1, \ldots, x_n \in \mathbb{Q}$ if $x_1 R_1 0, \ldots, x_n R_n 0$ then $k_1 x_1 + \ldots + k_n x_n + k_{n+1}\, Q\, 0$, where $R_1, \ldots, R_n, Q \in \{\geq, >\}$. Then $k_1 \geq 0, \ldots, k_{n+1} \geq 0$ and, if Q is $>$, then*

$$\left(\sum_{i=1, R_i \text{ is } >}^{n} k_i \right) + k_{n+1} > 0$$

Proof: We proceed by cases.
(Case 1) Let Q be $\geq$. Assume that $x_1 R_1 0, \ldots, x_n R_n 0$. Then, by hypothesis, we have that $k_{n+1} \geq -k_1 x_1 + \ldots - k_n x_n$. Suppose, by contradiction, that there exists $i \in \{1, \ldots, n\}$, such that $k_i < 0$. By suitable choices of the rational numbers $x_1, \ldots, x_n$, we can show that k_{n+1} is greater than every rational number. This is a contradiction with the hypothesis that $k_{n+1} \in \mathbb{Q}$. Therefore, $k_1 \geq 0, \ldots, k_n \geq 0$. Also, by suitable choices of the rational numbers $x_1, \ldots, x_n$, we can show that k_{n+1} is greater than every negative rational number. Therefore, $k_{n+1} \geq 0$.
(Case 2) Let now Q be $>$. Assume that $x_1 R_1 0, \ldots, x_n R_n 0$. Then, by hypothesis, we have that $k_{n+1} > -k_1 x_1 + \ldots - k_n x_n$. Similarly to the proof of (Case 1), we show that $k_1 \geq 0, \ldots, k_n \geq 0$. Without loss of generality, we can assume that, for some $m \geq 0$ and $m \leq n$, the first m relations $R_1, \ldots, R_m$ are $>$ and the subsequent relations $R_{m+1}, \ldots, R_n$ are $\geq$. If $m = 0$ then, by suitable choices of the rational numbers $x_1, \ldots, x_n$, we can show that k_{n+1} is greater than every nonpositive rational number. Therefore, $k_{n+1} > 0$. If $m > 0$ and $k_h > 0$, for some $h \in \{1, \ldots, m\}$, then, by suitable choices of the rational numbers $x_1, \ldots, x_n$, we can show that

k_{n+1} is greater than every negative rational number. Therefore, $k_{n+1} \geq 0$. Now, by joining the results of all possible cases for $h \in \{1, \ldots, m\}$, we obtain the following formula

$$\left(\bigvee_{h=1}^{m} \left(\bigwedge_{i=1, i \neq h}^{m} k_i \geq 0 \right) \wedge k_h > 0 \right) \wedge k_{m+1} \geq 0 \wedge \ldots \wedge k_{n+1} \geq 0$$

$$\vee \quad k_1 \geq 0 \wedge \ldots \wedge k_n \geq 0 \wedge k_{n+1} > 0$$

which is equivalent to the formula

$$k_1 \geq 0 \wedge \ldots \wedge k_{n+1} \geq 0 \wedge k_1 + \ldots + k_m + k_{n+1} > 0 \qquad \square$$

Theorem 23 *Let $t_1\, R_1\, 0, \ldots, t_n\, R_n\, 0$, and $u\, Q\, 0$ be atomic constraints such that $\mathcal{Q} \models \exists (t_1\, R_1\, 0 \wedge \ldots \wedge t_n\, R_n\, 0)$, where $R_i, Q \in \{\geq, >\}$ for $i = 1, \ldots, n$. Then $\mathcal{Q} \models \forall (t_1\, R_1\, 0 \wedge \ldots \wedge t_n\, R_n\, 0 \rightarrow u\, Q\, 0)$ iff there exist $k_1, \ldots, k_{n+1}$ such that (i) $k_1 t_1 + \ldots + k_n t_n + k_{n+1} = u$, (ii) $k_1 \geq 0, \ldots, k_{n+1} \geq 0$, and (iii) if Q is $>$ then*

$$\left(\sum_{i=1, R_i \text{ is } >}^{n} k_i \right) + k_{n+1} > 0$$

Proof: First, we consider the "if" part of the theorem. Let us assume that $k_1 t_1 + \ldots + k_n t_n + k_{n+1} = u$, for some $k_1 \geq 0, \ldots, k_{n+1} \geq 0$. The proof proceeds by cases.
(Case 1) Assume that Q is $\geq$. As a consequence, if $t_1\, R_1\, 0 \wedge \ldots \wedge t_n\, R_n\, 0$ where, for $i = 1, \ldots, n$, $R_i \in \{\geq, >\}$, then $k_1 t_1 + \ldots + k_n t_n + k_{n+1} \geq 0$.
(Case 2) Now, let us assume that Q is $>$, some $R_1, \ldots, R_m$ among the $R_1, \ldots, R_n$ are $>$, for $m \geq 0$ and $m \leq n$, and

$$\left(\sum_{i=1, R_i \text{ is } >}^{n} k_i \right) + k_{n+1} > 0$$

Then, either there exists $i \in \{1, \ldots, m\}$ such that $k_i > 0$, or $k_{n+1} > 0$. Therefore $k_1 t_1 + \ldots + k_n t_n + k_{n+1} > 0$.

Let us now consider the "only-if" part. Let us assume that $\mathcal{Q} \models \forall(t_1\, R_1\, 0 \wedge \ldots \wedge t_n\, R_n\, 0 \;\rightarrow\; u\, Q\, 0)$. Without loss of generality, we can also assume that the set $\{t_1 = 0, \ldots, t_m = 0\} \subseteq \{t_1 = 0, \ldots, t_n = 0\}$, with $m \leq n$, is a maximal set of linearly independent equations. Let us define the following affine transformation $\{X_1 = t_1, \ldots, X_m = t_m\}$, where every variable X_i, in the set $\{X_1, \ldots, X_m\}$ of variables of type `rat`, does not occur in the terms $t_1, \ldots, t_n, u$. By applying this transformation we infer that $\mathcal{Q} \models \forall(X_1\, R_1\, 0 \wedge \ldots \wedge X_m\, R_m\, 0 \wedge f_1(X_1, \ldots, X_m) R_{m+1} 0 \wedge \ldots \wedge f_{n-m}(X_1, \ldots, X_m) R_n 0 \;\rightarrow\; g(X_1, \ldots, X_m, V)\, Q\, 0)$, where the linear polynomials t_{m+1},...,t_n have been transformed into the linear polynomials $f_1(X_1, \ldots, X_m)$, ..., $f_{n-m}(X_1, \ldots, X_m)$, and the linear polynomial u has been transformed into the linear polynomial $g(X_1, \ldots, X_m, V)$, where $V = \mathit{vars}(u) - \mathit{vars}(\{t_1, ..., t_n\})$. Since $V \cap \{X_1, ..., X_m\} = \emptyset$, we have $\mathcal{Q} \models \forall\, (\; X_1\, R_1\, 0 \wedge \ldots \wedge X_m\, R_m\, 0 \wedge f_1(X_1, \ldots, X_m) R_{m+1} 0 \wedge \ldots \wedge f_{n-m}(X_1, \ldots, X_m) R_n 0 \rightarrow \forall\, V\; (g(X_1, \ldots, X_m, V)\, Q\, 0\;))$. Let us show that $V = \emptyset$. Suppose, by contradiction, that the set V is not empty. Without loss of generality we can assume that $g(X_1, \ldots, X_m, V)$ is of the form $aY + h(X_1, \ldots, X_m, V - \{Y\})$, where $a \neq 0$ and $Y \in V$, otherwise all the variables in V can be eliminated from $g(X_1, \ldots, X_m, V)$. As a consequence, the formula $\forall V\; (g(X_1, \ldots, X_m, V)\, Q\, 0)$ is equivalent to *false* in $\mathcal{Q}$, and this contradicts the hypothesis that $\mathcal{Q} \models \exists(t_1\, R_1\, 0 \wedge \ldots \wedge t_n\, R_n\, 0)$. Therefore, we can state that V is the empty set. Thus, we have $\mathcal{Q} \models \forall(X_1\, R_1\, 0 \wedge \ldots \wedge X_m\, R_m\, 0 \wedge f_1(X_1, \ldots, X_m) R_{m+1} 0 \wedge \ldots \wedge f_{n-m}(X_1, \ldots, X_m) R_n 0 \rightarrow g(X_1, \ldots, X_m)\, Q\, 0)$. A straightforward consequence is that $g(X_1, \ldots, X_m)$ is equivalent to

$$k_1 X_1 + \ldots + k_m X_m + k_{m+1} f_1(X_1, \ldots, X_m) + \ldots + k_n f_{n-m}(X_1, \ldots, X_m) + k_{n+1}$$

for some $k_1, \ldots, k_{n+1}$ (where $k_{m+1} = 0, \ldots, k_n = 0$). Hence, by Lemma 22, we have that $k_1 \geq 0$,...,$k_{n+1} \geq 0$ and if Q is $>$ then

$$\left(\textstyle\sum_{i=1, R_i \text{ is } >}^{n} k_i\right) + k_{n+1} > 0 \qquad \square$$

Note that the constraint matching procedure $\mathbf{CM_3}$ may generate *non-linear* polynomials. For instance, when looking for a substitution ϑ_2 such

that $\mathcal{Q} \models \forall(a \leftrightarrow \mu(a)\vartheta_2)$ (see Point 3.i above), an equivalence of the form $t \geq 0 \leftrightarrow u \geq 0$ will be rewritten as a conjunction of the form $nf(Vt-u) = 0, V > 0$, where $nf(Vt-u)$ is a suitably defined *normal form* of the nonlinear polynomial $Vt - u$ (see the rewriting rule (i) of Step 4). These nonlinear polynomials, however, will be of a particular form, which we call *bilinear* because the only nonlinear monomials that may occur are of the form cVX where c is a rational number, and V and X are distinct variables.

We now introduce bilinear polynomials and their normal forms.

Let t be a polynomial in the set $\{X_1, \ldots, X_n\}$ of variables and let $\{P_1, P_2\}$ be a partition of $\{X_1, \ldots, X_n\}$. t is *bilinear in* $\{P_1, P_2\}$ if: (i) for some grounding substitution σ_1 that acts as the identity substitution over the set P_2 of variables, $t\sigma_1$ is a linear polynomial, and (ii) for some grounding substitution σ_2 that acts as the identity substitution over the set P_1 of variables, $t\sigma_2$ is a linear polynomial. Let $X_1, \ldots, X_m$ (with $m \leq n$) be any ordering of the variables in P_1. A *normal form* $nf(t)$ of the bilinear polynomial t w.r.t. $X_1, \ldots, X_m$, is a polynomial of the form $a_1X_1 + \ldots + a_mX_m + a_{m+1}$, where $a_1, \ldots, a_{m+1}$ are linear polynomials in the set P_2 of variables and $\mathcal{Q} \models \forall\,(t = a_1X_1 + \ldots + a_mX_m + a_{m+1})$. Any linear polynomial t is also a bilinear polynomial in the partition $\{vars(t), \emptyset\}$, and thus, we can define a normal form $nf(t)$ w.r.t. any ordering of $vars(t)$.

Constraint Matching Phase: **CM$_3$**

Input: two clauses $\gamma' : H \leftarrow c \wedge B \wedge R$ and $\delta' : K \leftarrow d \wedge B$ in normal form.
Output: (a) a clause $\gamma'' : H \leftarrow e \wedge d\vartheta_2 \wedge B \wedge R$ such that: (i) $H \leftarrow c \wedge B \wedge R \cong H \leftarrow e \wedge d\vartheta_2 \wedge B \wedge R$, (ii) $B\vartheta_2 = B$, (iii) $vars(K\vartheta_2) \subseteq vars(H)$, and (iv) $vars(e) \subseteq vars(H, R)$, if such clause exists, and (b) FAILURE, otherwise.

In the following we denote the set $vars(c) - vars(B)$ by X, the set $vars(d) - vars(B)$ by Y, and the set $vars_c(B)$ by Z.

Step 1. Let the constraint e be the constraint $\tilde{e}$.

Step 2. Perform the following rewritings:

1. rewrite the constraints c and $d \wedge \tilde{e}$ into the equivalent non-redundant constraints $c'\colon t_1 R_1 0 \wedge \ldots \wedge t_m R_m 0$ and $d'\colon u_1 Q_1 0 \wedge \ldots \wedge u_n Q_n 0$, respectively;
2. rewrite c' and d' into the equivalent constraints $\overline{c}\colon nf(t_1)R_1 0 \wedge \ldots \wedge nf(t_m)R_m 0$ and $\overline{d}\colon nf(u_1)Q_1 0 \wedge \ldots \wedge nf(u_n)Q_n 0$, respectively, where, for $i=1,\ldots,m$ and $j=1,\ldots,n$, $nf(t_i)$ and $nf(u_j)$ are the normal forms of the linear polynomials t_i and u_j, respectively, w.r.t. the variable ordering $Z_1,\ldots,Z_h,Y_1,\ldots,Y_k,X_1,\ldots,X_l$, where $\{Z_1,\ldots,Z_h\}=Z$, $\{Y_1,\ldots,Y_k\}=Y$, and $\{X_1,\ldots,X_l\}=X$;

Step 3. Compute, if possible, an injective mapping μ from the atomic constraints in $\overline{c}$ to the atomic constraints in $\overline{d}$, such that if $\mu(t\,R\,0)$ is $u\,Q\,0$, then R and Q are the same predicate symbol. Let S be the set $\{t\,R\,0 \leftrightarrow u\,R\,0 \mid t\,R\,0 \in \overline{c}$ and $\mu(t\,R\,0)$ is $u\,R\,0\} \cup \{\overline{c} \rightarrow u\,Q\,0 \mid$ there is no $t\,R\,0 \in \overline{c}$ such that $\mu(t\,R\,0)$ is $u\,Q\,0\}$. If there exists no such injective mapping μ then return FAILURE;

Step 4. Let us consider a (sufficiently large) finite set W of variables of type `rat` such that $W \cap vars(\{\overline{c},\overline{d}\})=\emptyset$. Apply nondeterministically as long as possible the following rewriting rules to the set S:

(i) $\{t\,R\,0 \leftrightarrow u\,R\,0\} \cup S \Longrightarrow \{nf(Vt-u)=0, V>0\} \cup S$, where $V \in W$ and V does not occur in S;

(ii) $\{t_1 R_1 0 \wedge \ldots \wedge t_m R_m 0 \rightarrow u \geq 0\} \cup S \Longrightarrow \{nf(V_1 t_1 + \ldots + V_m t_m + V_{m+1} - u) = 0, V_1 \geq 0, \ldots, V_{m+1} \geq 0\} \cup S$, where $V_1,\ldots,V_{m+1} \in W$ and $V_1,\ldots,V_{m+1}$ do not occur in S;

(iii) $\{t_1 R_1 0 \wedge \ldots \wedge t_m R_m 0 \rightarrow u > 0\} \cup S \Longrightarrow \{nf(V_1 t_1 + \ldots + V_m t_m + V_{m+1} - u) = 0, V_1 \geq 0, \ldots, V_{m+1} \geq 0, \left(\sum_{i=1, R_i \text{ is} >}^{m} V_i\right) + V_{m+1} > 0\} \cup S$, where $V_1,\ldots,V_{m+1} \in W$ and $V_1,\ldots,V_{m+1}$ do not occur in S;

(iv) $\{c\,V + t = 0\} \cup S \Longrightarrow \{nf(c)=0, t=0\} \cup S$, if $V \in Z \cup X$;

(v) $\{cV + t = 0\} \cup S \Longrightarrow \{V = -\frac{t}{c}\} \cup S'$, if $V \in Y$, where $S' = \{nf(t)R0 \mid tR0 \in S\{V/-\frac{t}{c}\}\}$ and c is a rational number and c is different from 0;

(vi) $\{0 = 0\} \cup S \Longrightarrow S$;

(vii) $\{c = 0\} \cup S \Longrightarrow$ **fail**, if c is a rational number and c is different from 0;

where all the polynomials are bilinear in $\{X \cup Y \cup Z, W\}$ and the normal forms are computed w.r.t. the variable ordering $Z_1, \ldots, Z_h$, $Y_1, \ldots, Y_k$, $X_1, \ldots, X_l$. If S is rewritten to **fail**, then go to Step 3 looking for a new mapping μ;

Step 5. Let the set S computed at Step 4 be of the form $\{e_1, \ldots, e_p, u_1R'_10, \ldots, u_qR'_q0\}$, where, for $i = 1, \ldots, p$, e_i is an equation of the form $V = f(X, Y, W)$, for some variable $V \in Y$ and some linear polynomial $f(X, Y, W)$, and for $j = 1, \ldots, q$, $vars(u_j) \subseteq W$ and $R'_j \in \{\geq, >, =\}$. Let E be the set $\{e_1, \ldots, e_p\}$ and T be the set $\{u_1R'_10, \ldots, u_qR'_q0\}$ of linear equations and inequations. Generate from S the substitution ϑ_2 as follows:

1. if T is satisfiable, by using the Fourier-Motzkin algorithm [11] compute a solution of T of the form $\{W_1 = a_1, \ldots, W_s = a_s\}$, where for $i = 1, \ldots, s$, $W_i \in W$ and $a_i \in \mathbb{Q}$, and denote by σ_T the grounding substitution $\{W_1/a_1, \ldots, W_s/a_s\}$. Otherwise, if T is unsatisfiable, go to Step 3 looking for a new mapping μ;
2. let $\{V_1, \ldots, V_r\}$ be the set $\{V \in vars(u) \mid U \in Y,\ (U = u) \in E\sigma_T\} \cup (vars_c(K) - vars(d))$, let $\{b_1, \ldots, b_r\}$ be an arbitrary set of values in $\mathbb{Q}$, and let E' be $E\sigma_T\{V_1/b_1, \ldots, V_r/b_r\}$;
3. let ϑ_2 be the set $\{V/s \mid V = s \in E'\} \cup \{V_1/b_1, \ldots, V_r/b_r\}$;

Return the clause $\gamma'' \colon H \leftarrow e \wedge d\vartheta_2 \wedge B \wedge R$.

In the following theorems, we show that the constraint matching procedure $\mathbf{CM_3}$ is sound and complete with respect to its Input/Output specification. Thus, by using $\mathbf{CM_3}$ instead of $\mathbf{CM_1}$, we get a terminating, sound, and complete folding algorithm $\mathbf{FA_3}$ for applying Rule $\mathbf{Fold_3}$.

Theorem 24 (Soundness of the Constraint Matching Procedure $\mathbf{CM_3}$) *Let the two clauses $\gamma' : H \leftarrow c \wedge B \wedge R$ and $\delta' : K \leftarrow d \wedge B$ in normal form be the input of the constraint matching procedure* $\mathbf{CM_3}$. *If* $\mathbf{CM_3}$ *returns the clause $\gamma'' : H \leftarrow e \wedge d\vartheta_2 \wedge B \wedge R$, then: (i) $\gamma' \cong \gamma''$, (ii) $B\vartheta_2 = B$, (iii) $vars(K\vartheta_2) \subseteq vars(H)$, and (iv) $vars(e) \subseteq vars(H, R)$.*

Proof: Assume that the $\mathbf{CM_3}$ does not return FAILURE. We first show that: (ii) $B\vartheta_2 = B$. By definition, $vars_c(B) \subseteq Z$. We now show that $Z\vartheta_2 = Z$. By construction, at Step 3, $vars(S) \subseteq X \cup Y \cup Z$. At the end of Step 4, there is no formula in the set S that has an occurrence of a variable in Z. This is a consequence of the following facts. First, an application of rule (iv) produces the elimination of an occurrence in S of a variable in X or Z. Second, at the end of Step 4 we either obtain **fail** or a set S such that there is no occurrence in S of a variable in Z. This is due to the fact that every term in S is in normal form w.r.t. the variable ordering $Z_1, \ldots, Z_h, Y_1, \ldots, Y_k, X_1, \ldots, X_l$. Therefore, when in S there is no formula of the form $\varphi_1 \leftrightarrow \varphi_2$ or $\varphi_1 \rightarrow \varphi_2$, for some formulas φ_1 and φ_2, then we can only apply rule (iv), until there are no more variables in Z occurring in the set S of formulas. As a consequence, the substitution computed in the following Step 5 contains no variable in Z. Therefore, for any computed substitution ϑ_2 holds that $B\vartheta_2 = B$.

Let us now show that: (iii) $vars(K\vartheta_2) \subseteq vars(H)$. We first note that the variables in the set $vars_c(K) - vars(d)$ are bound, at Step 5 point 2, to ground terms of type `rat`. By hypothesis, the clause δ is in normal form. Therefore, if a variable occurs in d and in K then it does not occur in B. By definition, the set $vars(d) - vars(B)$ is denoted as Y. We want to show that the substitution ϑ_2 is of the form $\{V_1/t_1, \ldots, V_n/t_n\}$, where $\{V_1, \ldots, V_n\} = Y \cup vars_c(K) - vars(d)$, for some terms $t_1, \ldots t_n$ of type `rat` such that $vars(t_1, \ldots, t_n) \subseteq vars(H)$. At the end of Step 4, for every formula $\varphi \in S$, either φ is such that there is at least one occurrence in φ of a variable $V \in Y$ and φ is of the form $V = t$ for some term t, or φ contains only variables in W. Then, at Step 5, we distinguish the formulas in S into two sets: the set E of formulas of the form $Y = f(X, Y, W)$, for some linear polynomial $f(X, Y, W)$ and variable $V \in Y$, and the set T of formulas containing only

variables in W. At point 1, we compute a grounding substitution σ_T such that the formulas in the set T are satisfied. Then, at point 2, we compute a grounding substitution $\{V_1/b_1, \ldots, V_r/b_r\}$ where $V_1, \ldots, V_r$ are all the variables in Y that occur on the right hand-side of an equation in the set E. Therefore, the formulas in the set $E\sigma_T\{V_1/b_1, \ldots, V_r/b_r\}$ are of the form $V = g(X)$, for some linear polynomial $g(X)$ and variable $V \in Y$. By hypothesis, the clause γ is in normal form, therefore $X \subseteq vars(H)$.

It is straightforward to show that: (iv) $vars(e) \subseteq vars(H, R)$. It is enough to note that γ is in normal form and e is the constraint $\tilde{e}$.

We now show that: (i) $\gamma' \cong \gamma''$. Let us note that, by Lemma 15, the equivalence $H \leftarrow c \wedge B \wedge R \cong H \leftarrow e \wedge d\vartheta_2 \wedge B \wedge R$ holds iff $\mathcal{Q} \models \forall(c \leftrightarrow \exists W\,(e \wedge d\vartheta_2))$, where $W = vars(e \wedge d\vartheta_2) - vars(\{H, B\vartheta_2 \wedge R\})$. We proved that $vars(e) \subseteq vars(H, R)$, thus $W = vars(d\vartheta_2) - vars(\{H, B\vartheta_2 \wedge R\})$. Also, because the clause δ' is in normal form and the substitution ϑ_2 is such that $B\vartheta_2 = B$, we have that the clause $\delta'\vartheta_2$ is in normal form, therefore $vars(d\vartheta_2) \subseteq vars(\{K\vartheta_2, B\vartheta_2\})$. So, $W \subseteq vars(K\vartheta_2)$ and thus $W \subseteq vars(H)$, which is a contradiction. As a consequence, we deduce that W is the empty set, and the equivalence $H \leftarrow c \wedge B \wedge R \cong H \leftarrow e \wedge d\vartheta_2 \wedge B \wedge R$ holds iff $\mathcal{Q} \models \forall(c \leftrightarrow (e \wedge d\vartheta_2))$. In particular, we fix e to be $\tilde{e}$ and we want to show that $\mathbf{CM_3}$ computes a substitution ϑ_2 such that $\mathcal{Q} \models \forall(c \leftrightarrow (\tilde{e} \wedge d\vartheta_2))$.

At Step 2, we compute two non-redundant constraints $\overline{c}$ and $\overline{d}$ that are equivalent to the constraints c and $\tilde{e} \wedge d$, respectively. Then, at Step 3 we compute an injective mapping μ from the atomic constraints in $\overline{c}$ to the atomic constraints in $\overline{d}$. If we show that, for some substitution ϑ_2, $\mathcal{Q} \models \forall(\overline{c} \leftrightarrow \mu(\overline{c})\vartheta_2)$ and for every constraints $a \in (\overline{d} - \mu(\overline{c}))$, i.e. those constraints in $\overline{d}$ that do not belong to the image of the mapping μ, holds that $\mathcal{Q} \models \forall(\overline{c} \rightarrow a\vartheta_2)$, then the mapping μ is bijective from the set of atomic constraints in $\overline{c}$ to the set of atomic constraints in $\overline{d}\vartheta_2$. Thus, by Lemma 21 we have that $\mathcal{Q} \models \forall(c \leftrightarrow (\tilde{e} \wedge d\vartheta_2))$.

We want to show that if (1) the set S is $\{t_1 R_1 0 \wedge \ldots \wedge t_m R_m 0 \rightarrow uQ0\}$, (2) when Step 4 terminates S has not been rewritten into **fail**, and (3) at Step 5, point 1, we find a substitution σ_T such that the set T of inequalities is satisfied, then $\mathcal{Q} \models \forall(t_1 R_1 0 \wedge \ldots \wedge t_m R_m 0 \rightarrow uR0)$. Assume that Q is $\geq$, by rule (ii) we rewrite the set $\{t_1 R_1 0 \wedge \ldots \wedge t_m R_m 0 \rightarrow uQ0\}$ into the

set $\{nf(V_1t_1 + \ldots + V_mt_m + V_{m+1} - u) = 0, V_1 \geq 0, \ldots, V_{m+1} \geq 0\}$. By Theorem 23, if there exist $V_1, \ldots, V_{m+1}$ such that the set $\{nf(V_1t_1 + \ldots + V_mt_m + V_{m+1} - u) = 0, V_1 \geq 0, \ldots, V_{m+1} \geq 0\}$ of equalities and inequalities is satisfied, then we have $\mathcal{Q} \models \forall(t_1R_10 \wedge \ldots \wedge t_mR_m0 \rightarrow uR0)$. Let us now consider the rules (iv) and (v). Rule (iv) rewrites the set $\{cV + t = 0\}$, where V is a variable in $X \cup Z$, into the set $\{nf(c) = 0, t = 0\}$. We have that $\mathcal{Q} \models \forall(nf(c) = 0 \wedge t = 0 \rightarrow \forall V(cV + t = 0))$. Finally, by rule (v), we find a binding $\{V = -\frac{t}{c}\}$ for the variable $V \in Y$ such that $\mathcal{Q} \models \forall(cV + t = 0\{V/ - \frac{t}{c}\})$. Soundness of the rules (vi) and (vii) is straightforward. Moreover, the same argument provided for the soundness of rule (ii) can be used to prove the soundness of rule (iii), where Q is $>$. By noticing that the set $\{tR0 \leftrightarrow uR0\}$ can be rewritten into the set $\{tR0 \rightarrow uR0, uR0 \rightarrow tR0\}$, we can prove also that if S is $\{tR0 \leftrightarrow uR0\}$ and, when Step 4 terminates, S has not been rewritten into **fail**, and at Step 5, point 1, we find a substitution σ_T such that the set T of inequalities is satisfied, then $\mathcal{Q} \models \forall(tR0 \leftrightarrow uR0)$.

Finally, we show the soundness of Step 5. First, we note that the terms occurring in S are bilinear polynomials. Therefore, can always be rewritten into bilinear normal form. Therefore we can assume that polynomials in S are always in bilinear normal form. This property is invariant over the application of the rewriting rules (i),...,(vii). Therefore, when Step 4 terminates, if S has not been rewritten into **fail** then, at Step 5, the set W is a finite set of linear equalities and inequalities. Thus, the problem of finding a solution of T over $\mathbb{Q}$ is decidable and can be addressed by using the sound and complete Fourier-Motzkin algorithm [11], which is what we do at point 1. Also, at point 2, we compute a ground typed substitution for those variables $V \in Y$ that occur on the right hand-side of an equation of the form $U = u$, where $U \in Y$. This is sound because these variables are free and we can be substitute any ground term of type `rat` for them.

As a consequence of these observations we have that if $\mathbf{CM_3}$ terminates with an output different from FAILURE, then it computes a substitution ϑ_2 such that $\mathcal{Q} \models \forall(c \leftrightarrow (\tilde{e} \wedge d\vartheta_2))$. Thus, it computes a clause γ'' such that $\gamma' \cong \gamma''$. □

Theorem 25 (Completeness of the Constraint Matching Procedure $\mathbf{CM_3}$) *The constraint matching procedure* $\mathbf{CM_3}$ *terminates for all input clauses. Moreover, if the input clauses are* $\gamma' : H \leftarrow c \wedge B \wedge R$ *and* $\delta' : K \leftarrow d \wedge B$ *in normal form and it is possible to find a constraint* e *and a substitution* ϑ_2 *such that: (i)* $H \leftarrow c \wedge B \wedge R \cong H \leftarrow e \wedge d\vartheta_2 \wedge B \wedge R$, *(ii)* $B\vartheta_2 = B$, *(iii)* $vars(K\vartheta_2) \subseteq vars(H)$, *and (iv)* $vars(e) \subseteq vars(H, R)$, *then the output of* $\mathbf{CM_3}$ *is not FAILURE.*

Proof: We start by proving that $\mathbf{CM_3}$ terminates for all input clauses. Step 1 terminates because consists of a single application of the *solve* function. Step 2 transforms, by a finite number of terminating steps, the constraints c and $\tilde{e} \wedge d$ into equivalent non-redundant constraints in normal form. Step 3 performs the computation of an injective mapping from a finite set to a finite set. This is an always terminating operation. Also, the number of possible mappings is finite. Let us then consider Step 4. Rules (i), (ii), and (iii) can be applied only a finite number of times, as the number of formulas in S of the form $a \rightarrow b$ or $a \leftrightarrow b$ is finite and is decreased by 1 at each application of the rules (i), (ii), and (iii). Let us now consider rule (v). This rule replaces an equation of the form $cV + t = 0$ in S, where $V \in Y$, by an equation of the form $V = -\frac{t}{c}$. Moreover, it replaces every other occurrence of V in S by the term $-\frac{t}{c}$ which does not contain V. As a consequence, no rule can be further applied to the equation $V = -\frac{t}{c}$, and all the remaining occurrences of V in S are eliminated. Note that, by construction, the term c is a ground term of type `rat`. Thus, the application of the rule (v) can increase the number of occurrences of the variables in X, Y, or Z but only a finite number of times, as the number of occurrences in S of variables in Y is finite. Finally, let us consider the rule (iv). This rule eliminates an occurrence in S of a variable in $X \cup Z$. As we noted previously, the number of occurrences in S of the variables in $X \cup Z$ can increase, due to the application of rule (v). However, this can happen only a finite number of time. Therefore, as the application of rule (iv) decreases the number of occurrences in S of the variables in $X \cup Z$, we can deduce that the rewriting procedure of Step 4 terminates, for every given finite set S. The operations performed in Step 5 terminates, due to

the termination of the Fourier-Motzkin algorithm. Finally, we point out that the both from Step 4 and Step 5 we can return to Step 3 in order to compute a different mapping μ. As previously said, the number of possible injective mappings μ from a finite set to a finite set are finite. Therefore, $\mathbf{CM_3}$ terminates for all input clauses.

Assume that there exists a constraint e and a substitution ϑ_2 such that $H \leftarrow c \wedge B \wedge R \cong H \leftarrow e \wedge d\vartheta_2 \wedge B \wedge R$ and (i) $B\vartheta_2 = B$, (ii) $vars(K\vartheta_2) \subseteq vars(H)$, and (iii) $vars(e) \subseteq vars(H, R)$. We want to show that $\mathbf{CM_3}$ does not return FAILURE. By Lemma 20, we can define e as the constraint $\tilde{e}$ at Step 1 and be complete. At Step 2 we rewrite the constraints c and $\tilde{e} \wedge d$ into the equivalent constrains $\overline{c}$ and $\overline{d}$. By the assumptions and these observations, follows that exists a substitution ϑ_2 such that $\mathcal{Q} \models \forall(\overline{c} \leftrightarrow (\overline{d}\vartheta_2))$. The constraints $\overline{c}$ and $\overline{d}$ are non-redundant, but we do not know if also the constraint $\overline{d}\vartheta_2$ is non-redundant. It is always possible to select a subset of the atomic constraints in $\overline{d}\vartheta_2$ such that their conjunction d' is a non-redundant constraint equivalent to $\overline{d}\vartheta_2$. Thus, by Lemma 21, follows that there exists a bijection μ from the atomic constraints in $\overline{c}$ to the atomic constraints in d' such that the constraints are equivalent pairwise. However, we still have to compute the substitution ϑ_2. Therefore, we compute an injective mapping μ from the atomic constraints in $\overline{c}$ to the atomic constraints in $\overline{d}$ and compute a substitution ϑ_2 such that $\overline{c}$ is equivalent to $\mu(\overline{c})\vartheta_2$ and $\overline{c}$ implies every constraint in $(\overline{d} - \mu(\overline{c}))\vartheta_2$. This is complete, because by backtracking we compute all possible injective mappings μ. We have to show that, given the injective mapping μ, if exists a substitution ϑ_2 such that $\overline{c}$ is equivalent to $\mu(\overline{c})\vartheta_2$ and $\overline{c}$ implies every constraint in $(\overline{d} - \mu(\overline{c}))\vartheta_2$ then Steps 4 and 5 do not return FAILURE. Let us consider rule (i): if exists a substitution ϑ_2 such that $tR0$ is equivalent to $(uR0)\vartheta_2$ then, by Theorem 23, it is possible to find a substitution that satisfies the set of equalities and inequalities $\{nf(Vt - u) = 0, V > 0\}$. Indeed, we can rewrite the set $\{tR0 \leftrightarrow uR0\}$ into the set $\{tR0 \rightarrow uR0, uR0 \rightarrow tR0\}$ and, by Theorem 23, we have that if $\forall(tR0 \rightarrow uR0)$ and $\forall(uR0 \rightarrow tR0)$ hold then it is possible to find ground typed substitutions for the variables V_1, V_2, V_3, and V_4 of type `rat` such that the set $\{u = V_1t + V_2, V_1 \geq 0, V_2 \geq 0, t = V_3u + V_4, V_3 \geq 0, V_4 \geq 0\}$ of

equations and inequations is satisfied, where without loss of generality we have assumed that R is $\geq$. By standard algebraic manipulation, we can obtain an equivalent set $\{V_1t+V_2-u=0, V_1>0, V_2=0, V_3y+V_4-t=0, V_3>0, V_4=0\}$ of equations and inequations. This can be further simplified into $\{V_1t-u=0, V_1>0\}$ which, modulo bilinear normal form, is the output of the rewriting rule (i). By similar observation we can prove the case where R is $>$. Furthermore, by Theorem 23 and similar observations, we can motivate also completeness for rules (ii) and (iii). Moreover, if we have that for all V the equality $cV+t=0$ holds, then the set $\{nf(c)=0, t=0\}$ of equations can be satisfied. Thus, rule (iv) is complete. Similar observations can be done for rules (v), (vi), and (vii). Thus, we deduce that Step 4 does not return FAILURE. Proving the completeness of Step 5 is trivial, as it relies on the completeness of the Fourier-Motzkin algorithm. □

Folding Algorithm: **FA$_3$**

Input: two clauses $\gamma\colon H \leftarrow c \wedge G$ and $\delta\colon K \leftarrow d \wedge B$ in normal form and with no common variables;

Output: the clause $\eta\colon H \leftarrow e \wedge K\vartheta \wedge R$, if it is possible to fold γ using δ according to the Definition 10, and FAILURE otherwise.

Step 1. Execute the goal matching procedure **GM**, presented in Section 4.2 with the clauses γ and δ as input. If the output of **GM** is FAILURE then return FAILURE else let the output of **GM** be the two clauses $\gamma'\colon H \leftarrow c \wedge B\vartheta_1 \wedge R$ and $\delta'\colon K\vartheta_1 \leftarrow d\vartheta_1 \wedge B\vartheta_1$;

Step 2. Execute the constraint matching procedure **CM$_3$** with the clauses γ' and δ' as input. If the output of **CM$_3$** is FAILURE then go to Step 1 and search for a new pair of output clauses of **GM**;

Step 3. If no output of **GM** can be found such that the output of **CM$_3$** is different from FAILURE, then return FAILURE else let the output of **CM$_3$** be the clause $\gamma''\colon H \leftarrow e \wedge d\vartheta_1\vartheta_2 \wedge B\vartheta_1 \wedge R$ and let ϑ be the substitution $\vartheta_1\vartheta_2$. Return $\eta\colon H \leftarrow e \wedge K\vartheta \wedge R$.

Theorem 26 (Soundness of the Folding Algorithm FA$_3$) *Let the two clauses $\gamma\colon H \leftarrow c \wedge G$ and $\delta\colon K \leftarrow d \wedge B$, in normal form and without common variables, be the input of the Folding Algorithm* **FA$_3$**. *If the Folding Algorithm* **FA$_3$** *returns the clause $\eta\colon H \leftarrow e \wedge K\vartheta \wedge R$, then the clause η can be obtained by folding γ using δ according to Definition 10.*

Proof: If the two clauses $\gamma\colon H \leftarrow c \wedge G$ and $\delta\colon K \leftarrow d \wedge B$, in normal form and without common variables are the input of the Folding Algorithm **FA$_3$** then by the Soundness Theorem 13 we have that, at Step 1, $H \leftarrow c \wedge G \cong H \leftarrow c \wedge B\vartheta_1 \wedge G$ and $H \leftarrow c \wedge B\vartheta_1 \wedge G$ is in normal form. Also, by the Soundness Theorem 24, at Step 2 we have $H \leftarrow c \wedge B\vartheta_1 \wedge G \cong H \leftarrow e \wedge d\vartheta_1\vartheta_2 \wedge B\vartheta_1 \wedge G$. By Theorem 24, we have that $B\vartheta_1\vartheta_2 = B\vartheta_1$. Let the substitution ϑ be the substitution $\vartheta_1\vartheta_2$, then we have $H \leftarrow c \wedge G \cong H \leftarrow e \wedge d\vartheta \wedge B\vartheta \wedge G$.

Let X be a variable in $evars(\delta)$, by Theorem 13 we have that $X\vartheta_1$ is a variable and it does not occur in $Y\vartheta_1$, for every variable Y occurring in $d \wedge B$ and different from X. We have also that δ' is in normal form. Therefore, $X\vartheta_1 \in vars(B\vartheta_1)$. By Theorem 24, $B\vartheta_1\vartheta_2 = B\vartheta_1$ and so $X\vartheta_1\vartheta_2 = X\vartheta_1$. That is, $X\vartheta_1\vartheta_2$ is a variable. Also, $X\vartheta_1 \notin vars(\{H, R\})$ and $vars(e) \subseteq vars(\{H, R\})$. Thus, $X\vartheta_1\vartheta_2 \notin vars(\{H, e, R\})$. Let us now show that $X\vartheta_1\vartheta_2$ does not occur in $Y\vartheta_1\vartheta_2$. By Theorem 24, if $Y\vartheta_1\vartheta_2 \neq Y\vartheta_1$ then for all variables $V \in vars(Y\vartheta_1)$ such that $V\vartheta_2 \neq V$ we have $vars(V\vartheta_2) \subseteq vars(H)$, because δ' is in normal form, $B\vartheta_1\vartheta_2 = B\vartheta_1$, and $vars(K\vartheta_1\vartheta_2) \subseteq vars(H)$. As a consequence, $X\vartheta_1\vartheta_2$ does not occur in $Y\vartheta_1\vartheta_2$, for every variable Y occurring in $d \wedge B$ and different from X, because $\{X\vartheta_1\vartheta_2 \mid X \in evars(\delta)\} \cap vars(H) = \emptyset$. Finally, by Theorem 24, $K\vartheta_1\vartheta_2 \subseteq vars(H)$. □

Theorem 27 (Completeness of the Folding Algorithm FA$_3$) *Let the two clauses $\gamma : H \leftarrow c \wedge G$ and $\delta : K \leftarrow d \wedge B$, in normal form and with no common variables, be the input of the Folding Algorithm* **FA$_3$**. *The Folding Algorithm* **FA$_3$** *terminates. Moreover, if it is possible to fold γ using δ according to Definition 10, then the output of the Folding Algorithm* **FA$_3$** *is a clause η, that is the output is not FAILURE.*

Proof: By Theorems 14 and 25, Step 1 and Step 2 of the Folding Algorithm **FA$_3$** terminate. At Step 3 we might have to backtrack to Step 1 in

order to compute a different clause γ'. However, for a given pair of clauses γ and δ, the number of possible different clauses γ' computed by the goal matching procedure **GM** is finite. Therefore, the Folding Algorithm $\mathbf{FA_3}$ terminates.

Let us assume that there exist a constraint e, a goal R and a substitution ϑ such that Conditions 1 and 2 of Definition 10 are satisfied. Assume also that, given the clauses γ and δ as input, the Folding Algorithm $\mathbf{FA_2}$ returns FAILURE. Then, either the Goal Matching Phase returns FAILURE or, for every clause γ' computed by the goal matching procedure **GM**, the constraint matching procedure $\mathbf{CM_3}$ returns FAILURE. The first case is impossible because, by Theorem 14 and under the given hypotheses, **GM** does not return FAILURE. In the second case, by the transitivity of $\cong$, we have that $H \leftarrow c \wedge B\vartheta_1 \wedge R \cong H \leftarrow e \wedge d\vartheta \wedge B\vartheta \wedge R$. As a consequence, we have that $B\vartheta_1 = B\vartheta$. By hypothesis, we have also that $vars(K\vartheta) \subseteq vars(H)$. Therefore, either $vars(e) \subseteq vars(\{H, R\})$ or we can eliminate the variables in the set $vars(e) - vars(\{H, R\})$ by using the function solve. Indeed, by the properties of the substitution ϑ and by the fact that $vars(K\vartheta) \subseteq vars(H)$, we have that $vars(e) \cap vars(d\vartheta \wedge B\vartheta) = \emptyset$. Under these assumptions, the Completeness Theorem 25 states that the output of $\mathbf{CM_3}$ is different from FAILURE. Thus, the Folding Algorithm $\mathbf{FA_2}$ does not return FAILURE. □

Let us not illustrate the folding algorithm $\mathbf{FA_3}$ by considering the third example proposed at the beginning of this chapter. We are given the following clauses as input:

γ: $p(X_1, X_2) \leftarrow Z > 0 \wedge X_1 \geq Z+1 \wedge X_2 > X_1 \wedge s(Z)$
δ: $q(Y_1, Y_2) \leftarrow W > 0 \wedge Y_1 - 3 \geq 2W \wedge 2Y_2 > Y_1 + 1 \wedge s(W)$

These clauses are in normal form and do not share any variable. The goal matching procedure **GM**, applied to the clauses γ and δ, returns the following pair of clauses:

γ': $p(X_1, X_2) \leftarrow Z > 0 \wedge X_1 \geq Z+1 \wedge X_2 > X_1 \wedge s(Z)$
δ': $q(Y_1, Y_2) \leftarrow W > 0 \wedge Y_1 - 3 \geq 2W \wedge 2Y_2 > Y_1 + 1 \wedge s(Z)$

Let us now consider an execution of the constraint matching procedure

CM$_3$, applied to the clauses γ' and δ'. We start off by computing, at Step 1, the following constraint e which is defined as $\mathit{solve}(Z > 0 \wedge X_1 \geq Z{+}1 \wedge X_2 > X_1,\ \{X_1, X_2\})$:

$e\colon\ X_1 \geq 1 \wedge X_2 > X_1$

By performing the rewritings listed at Step 2, and choosing a suitable injective mapping as described at Step 3, we get the following set:

$$S\colon\ \{Z > 0 \leftrightarrow Z > 0,\ -Z + X_1 - 1 \geq 0 \leftrightarrow -2Z + Y_1 - 3 \geq 0,\\ X_2 - X_1 > 0 \leftrightarrow 2Y_2 - Y_1 - 1 > 0\}$$

By applying the rewriting rules listed at Step 4, we rewrite S into a new set which, by using the notation introduced at the beginning of Step 5, can be written as $E \cup T$, where:

$E\colon\ \{Y_1 = W_2X_1 + 3 - W_2,\ Y_2 = (\frac{W_2}{2} - \frac{W_3}{2})X_1 + \frac{W_3}{2}X_2 + 2 - \frac{W_2}{2}\}$

$T\colon\ \{W_1 - 1 = 0,\ W_1 > 0,\ 2 - W_2 = 0,\ W_2 > 0,\ W_3 > 0\}$

At Point 1 of Step 5 we compute the substitution $\sigma_T = \{W_1/1,\ W_2/2,\ W_3/2\}$, which is a solution of the set T of equations and inequations. Therefore, at Points 2 and 3 of Step 5 we compute the following substitution:

$\vartheta_2\colon\ \{Y_1/2X_1 + 1,\ Y_2/X_2 + 1\}$

Finally, the output of the folding algorithm **FA$_3$** is the clause $p(X_1, X_2) \leftarrow e \wedge q(Y_1, Y_2)\vartheta_2$, that is:

$\eta\colon\ p(X_1, X_2) \leftarrow X_1 \geq 1 \wedge X_2 > X_1 \wedge q(2X_1{+}1, X_2{+}1)$

Clause η has no existential variables.

4.5 Related Work and Conclusions

The folding rule has been considered in several papers that deal with the transformation rules for logic programs and constraint logic programs [9, 24, 29, 45, 79]. Folding is a crucial transformation rule because, besides other reasons, it allows the change of the recursive structure of the programs and, when using program transformation for inductive proofs of program properties [57], its application is basically equivalent to the use of an inductive hypothesis.

In the literature the folding rule is specified in a declarative way and no algorithm is provided to determine whether or not, given a clause γ to be folded and a clause δ for folding, one can actually fold γ using δ.

In this chapter we have considered constraint logic programs with constraints that are conjunctions of linear equations and inequations over the rational numbers (or the real numbers) and we have proposed an algorithm, based on linear algebra and term rewriting techniques, for applying the folding rule.

We have also introduced two variants of the folding rule and we have presented two algorithms for applying these variants. The first variant combines the folding rule with the clause splitting rule and the second variant can be applied for eliminating the existential variables of a clause, that is, the variables which occur in the body of a clause and not in the head.

As already pointed out in Section 4.2, the problem of folding a clause γ using a clause δ is related to the problem of *matching* two terms modulo an equational theory [15, 81]. In particular, if γ is $H \leftarrow c \wedge G$ and δ is $K \leftarrow d \wedge B$ then the problem of computing a constraint e, a substitution ϑ and a goal R such that $H \leftarrow c \wedge G \cong H \leftarrow e \wedge (d \wedge B)\vartheta \wedge R$ can be seen as the problem of computing a substitution σ such that $H \leftarrow c \wedge G =_T (H \leftarrow X \wedge d \wedge B \wedge Y)\sigma$ where: (i) X and Y are typed variables ranging on constraints and goals, respectively, (ii) the substitution σ is such that $X\sigma = e$, $Y\sigma = R$, $(d \wedge B)\sigma = (d \wedge B)\vartheta$, and (iii) the equality $=_T$ holds modulo the theory T of the constraints and the goals. However, there are relevant differences between the folding problem and the matching problem: the requirement of a typed substitution and the extra conditions on the existential variables occurring in e and in R and on the substitution σ deriving from the definitions of the **Fold$_1$**, **Fold$_2$**, and **Fold$_3$** rules. This extension of the matching problem with conditions on existential variables and typing has not been considered in the literature and existing methods cannot be directly applied to our case.

In our approach we have divided the general problem of folding into the two problems of goal matching and constraint matching. Then, we have presented the goal matching procedure **GM** which is a complete procedure

to solve the problem of matching modulo the $\mathrm{AC}_\wedge$ theory [42, 77] tailored to the generation of substitutions that satisfy our additional requirements on existential variables. In order to solve the goal matching problem, it is also possible to use standard AC-matching algorithms such as the one presented in [23] and then discard those solutions that do not satisfy our additional requirements. However, as already mentioned in Section 4.2, the AC-matching problem is NP-complete and therefore we are limited by this theoretical bound.

For the problem of constraint matching we need to distinguish the three cases corresponding to the three proposed folding rules. In the case of **Fold$_1$** and **Fold$_2$** we have to verify whether $\mathcal{Q} \models \forall(c \leftrightarrow g)$ and $\mathcal{Q} \models \forall(c \leftrightarrow g_1 \vee \cdots \vee g_n)$, respectively, where g is the constraint obtained from $e \wedge d$, and $g_1, \ldots, g_n$ are the constraints obtained from $e_1 \wedge d, \ldots, e_n \wedge d$, respectively, by eliminating the local variables and the constraints $c, d, e, e_1, \ldots, e_n$ are given. Since the theory of constraints we consider admits quantifier elimination [44], these two problems can be solved by using existing quantifier elimination algorithms [11], which indeed has been our approach in this work.

In the case of **Fold$_3$**, we have to compute a substitution ϑ_2 such that $\mathcal{Q} \models \forall(c \leftrightarrow e \wedge d\vartheta_2)$, where the constraints c, d, and e are given. No obvious solution using the quantifier elimination technique can be provided for the solution of this problem, because of the need of computing the substitution ϑ_2. However, the problem of computing the substitution ϑ_2 can be seen as a generalized form of matching problem modulo the equational theory Q of the constraints, where the substitution ϑ is not applied to the whole right hand-side but to a subterm of it. In particular, recalling that by the conditions in **Fold$_3$** we have $\mathit{vars}(e) \subseteq \mathit{vars}(c)$, the problem can be seen as the following *restricted unification* problem [16]: given two constraints c and $e \wedge d$ find a substitution ϑ_2 such that $(c =_Q e \wedge d)\vartheta_2$, where ϑ_2 is restricted to the set $\mathit{vars}(d) - \mathit{vars}(c)$ of variables. In [16] it is described how to obtain, under certain conditions over the equational theory, an algorithm for solving a restricted unification problem from an algorithm that solves the corresponding unrestricted unification problem.

To the best of our knowledge, for the considered theory Q of constraints

there is no *ad-hoc* solution neither for the restricted unification problem nor for the unrestricted unification problem. However, in the literature several *combination* methods have been proposed [7, 64, 72]. These methods consist in considering a given theory as the union of simpler theories for which it is known an algorithm that solves the problem of interest (unification, in our case): the theory of constraints Q, in particular, can be seen as the union of the theory AG of *abelian groups*, BA of *boolean algebras*, PO of *partial orders* [6], and some additional axioms whose symbols belong to different signatures of the component theories. An example of this latter kind of axioms is the following: $X \geq Y =_Q X + Z \geq Y + Z$, which expresses the monotonicity of addition w.r.t. the $\geq$ ordering relation. Once that the theory of interest has been expressed as the union of known theories, these combination methods allow, under some conditions on the component theories, to obtain a unification algorithm for the union-theory by combining the corresponding algorithms developed for the component theories. In the case of the theory Q, however, we cannot use the combination methods proposed in the literature so far, since our theory does not satisfy certain syntactic conditions which ensure the applicability of these methods. In particular, the theory Q is neither *collapse-free* nor *regular* and the signatures of the component theories are not disjoint (see [6] for a definition of these notions). This motivates our construction of an *ad-hoc* algorithm for solving the constraint matching problem.

An issue left for further research is the development of a complete folding algorithm for the $\mathbf{Fold_1}$ rule, which is the standard folding rule. The source of incompleteness in our algorithm is the $\mathbf{CM_1}$ procedure, that looks for a solution to the constraint matching problem. We have defined the formula $\hat{e}$ which is the most general formula such that the clause equivalence $H \leftarrow c \wedge B \wedge R \cong H \leftarrow \hat{e} \wedge d \wedge B \wedge R$ holds, but the formula $\hat{e}$ may in general be not a constraint. The $\mathbf{CM_1}$ procedure computes a formula $e_1 \vee \ldots \vee e_n$ which is equivalent to the formula $\hat{e}$ and is in disjunctive normal form. If there is a constraint $e \in \{e_1, \ldots, e_n\}$ such that the clause equivalence $H \leftarrow c \wedge B \wedge R \cong H \leftarrow e \wedge d \wedge B \wedge R$ holds then e is a solution of the constraint matching problem, else we do not know whether such a constraint e exists or not. In a geometrical interpretation, the problem of computing

the constraint e corresponds to the problem of finding, if it exists, a convex polytope e contained in the (possibly) non convex polytope $\hat{e}$ and such that the above clause equivalence holds. The problem of computing the convex e by geometrical methods is not trivial because the set of convexes contained in $\hat{e}$ may be infinite and this makes it unfeasible to generate this convexes and test whether there is one that satisfies our requirements.

Another task for future work is the adaptation of our folding algorithms to other constraint domains, such as the linear equations and inequations over the integer numbers.

One important aspect that needs to be addressed is the analysis of the computational complexity of our algorithms for folding. As already mentioned, the **GM** goal matching procedure is of NP time complexity in the size of the input formulas. The $\mathbf{CM_1}$, $\mathbf{CM_2}$, and $\mathbf{CM_3}$ procedures make use of the Fourier-Motzkin variable elimination algorithm, which in the worst case takes superexponential time in the number of variables occurring in the input clauses. However, there are more efficient algorithms. In particular, in the $\mathbf{CM_1}$ and $\mathbf{CM_2}$ procedures we need to perform quantifier elimination from a linear formula of the theory of ordered fields of reals with bounded quantifier alternation. For this kind of quantifier elimination problem there exist algorithms of exponential time complexity [82]. In the $\mathbf{CM_3}$ procedure we need to test for the satisfiability of linear formulas of the theory of ordered fields of reals. For this kind of satisfiability test there exist algorithms of polynomial time complexity [34]. We leave it for future research to make a tighter complexity analysis of our algorithms. It is interesting to notice, though, that in practice the Fourier-Motzkin algorithm tend to perform better that the above mentioned methods. This motivates the fact that many implementations use the Fourier-Motzkin algorithm.

Finally, an implementation of our folding algorithms is under development in the MAP transformation system [46]. This implementation will allow us to evaluate in practice the efficiency of the folding algorithms, as well as the usefulness of the various versions of the folding rule in various program derivation techniques.

Chapter 5

Transformational Verification of Parameterized Protocols Using Array Formulas

Protocols are rules that govern the interactions among concurrent processes. In order to guarantee that these interactions enjoy some desirable properties, many sophisticated protocols have been designed and proposed in the literature. These protocols are, in general, difficult to verify because of their complexity and ingenuity. This difficulty has motivated the development of methods for the formal specification and the automated verification of properties of protocols. One of the most successful methods is *model checking* [18]. It can be applied to any protocol that can be formalized as a *finite state system*, that is, a finite set of transitions over a finite set of states.

Usually, the number of interacting concurrent processes is not known in advance. Thus, people have designed protocols that can work properly for any number of interacting processes. These protocols are said to be *parameterized* with respect to the number of processes. Several extensions of the model checking technique based upon *abstraction* and *induction* have been

proposed in the literature for the verification of parameterized protocols (see, for instance, [5, 48, 76, 83]). However, since the general problem of verifying temporal properties of parameterized protocols is undecidable [4], these extensions cannot be fully mechanical.

In this chapter we propose an alternative verification method based on *program transformation* [17]. Our main objective is to establish a correspondence between protocol verification and program transformation, so that the large number of semi-automatic techniques developed in the field of program transformation can be applied to the verification of properties of parameterized protocols.

Since arrays are often used in the design of parameterized protocols, we will consider a specification language that allows us to write *array formulas*, that is, first order formulas over arrays. We will specify a parameterized protocol and a property of interest by means of a logic program whose clause bodies may contain array formulas. Our verification method works by transforming this logic program, in which we assume that the head of the clause specifying the property has predicate *prop*, into a new logic program where the clause $prop \leftarrow$ occurs. Our verification method is an extension of many other techniques based on logic programming which have been proposed in the literature [22, 26, 30, 41, 50, 63, 69].

We will demonstrate our method by considering the parameterized Peterson's protocol [51]. This protocol ensures mutually exclusive use of a given resource which is shared among N processes. The number N is the parameter of the parameterized protocol. In order to formally show that Peterson's protocol ensures mutual exclusion, we cannot use the model checking technique directly. Indeed, since the parameter N is unbounded, the parameterized Peterson's protocol, as it stands, cannot be viewed as a finite state system. Now, one can reduce it to a finite state system, thereby enabling the application of model checking, by using the above mentioned techniques based on abstraction [5]. However, it is not easy to find a powerful abstraction function which works for the many protocols and concurrent systems one encounters in practice.

In contrast, our verification method based on program transformation does not rely on an abstraction function which is applied once at the be-

ginning of the verification process, but it relies, instead, on a *generalization strategy* which is applied *on demand* during the construction of the proof, possibly many times, depending on the structure of the portion of proof constructed so far. This technique provides a more flexible approach to the problem of proving properties of protocols with an infinite state space.

The chapter is structured as follows. In Section 5.1 we recall the parameterized Peterson's protocol for mutual exclusion which will be used throughout the chapter as a working example. In Section 5.2 we present our specification method which makes use of an extension of stratified logic programs where bodies of clauses may contain first order formulas over arrays of parameterized length. We consider properties of parameterized protocols that can be expressed by using formulas of the branching time temporal logic CTL [18] and we show how these properties can be encoded by stratified logic programs with array formulas. Then, in Section 5.3, we show how CTL properties can be proved by applying unfold/fold transformation rules to a given specification. In Section 5.4 we discuss some issues regarding the automation of our transformation method. Finally, in Section 5.5 we briefly discuss the related work in the area of the verification of parameterized protocols.

In Appendix C we give a further demonstration of our verification method by using it to prove a collision avoidance property of the Token Ring protocol.

5.1 Peterson's mutual exclusion protocol

In this section we provide a detailed description of the parameterized Peterson's protocol [51]. The goal of this protocol is to ensure the mutually exclusive access to a resource that is shared among N (≥ 2) processes. Let assume that for any i, with $1 \leq i \leq N$, process i consists of an infinite loop whose body is made out of two portions of code: (i) a portion called *critical section*, denoted *cs*, in which the process uses the resource, and (ii) a portion called *non-critical section*, denoted *ncs*, in which the process does not use the resource. We also assume that every process is initially in its non-critical section.

We want to establish the following *Mutual Exclusion* property of the computation of the given system of N processes: *for all i and j in $\{1, \ldots, N\}$, while process i executes a statement of its critical section, process j, with $j \neq i$, does not execute any statement of its critical section.*

The parameterized Peterson's protocol consists in adding two portions of code to every process: (i) a first portion to be executed before entering the critical section, and (ii) a second portion to be executed after exiting the critical section (see in Figure 5.1 the code relative to process i).

Peterson's protocol makes use of two arrays $Q[1, \ldots, N]$ and $S[1, \ldots, N]$ of natural numbers, which are shared among the N processes. The N elements of the array Q may get values from 0 to $N-1$ and are initially set to 0. The N elements of the array S may get values from 1 to N and their initial values are not significant (in [51] it is assumed that they are all 1's). Notice that in [51] the array S is assumed to have $N-1$ elements, not N as we do. Indeed, the last element $S[N]$ is never used by Peterson's protocol. Its introduction, however, allows us to write formulas which are much simpler. Notice also that, in the original paper, in place of the while-loop (see Figure 5.1) it was used a for-loop. However, our formulation is equivalent and allows for a more intuitive encoding of the loop as a state transition system.

In Peterson's protocol we also have the array $J[1, \ldots, N]$ whose i-th element, for $i = 1, \ldots, N$, is a local variable of process i and may get values from 1 to N. Notice that the array J is *not* shared and indeed, for $i = 1, \ldots, N$, process i reads and/or writes $J[i]$ only.

In Figure 5.2 process i is represented by a finite state diagram. In that diagram a transition from state a to state b is denoted by an arrow from a to b labelled by a test t and a statement s. We have omitted from the label of a transition the test t when it is *true*. Likewise, we have omitted the statement s when it is *skip*. A transition is said to be *enabled* iff its test t evaluates to *true*. An enabled transition takes place by executing its statement s.

For $i = 1, \ldots, N$, process i is deterministic in the sense that in any of its states at most one transition is enabled. However, in the given system of N processes, it may be the case that more than one transition is enabled

(obviously, no two enabled transitions belong to the same process). In that case we assume that exactly one of the enabled transitions takes place. Note that we do not make any fairness assumption so that, for instance, if the same configuration of enabled transitions occurs again in the future, nothing can be said about the transition which will actually take place in that repeated configuration.

---

while $true$ **do**
  ncs : non-critical section of process i;
    $J[i] := 1$;
  w: **while** $J[i] < N$ **do**
    $Q[i] := J[i]$; $S[J[i]] := i$;
  λ: **await** $\forall k\,(k \neq i \rightarrow Q[k] < J[i]) \vee (S[J[i]] \neq i)$;
    $J[i] := J[i]+1$
  od;
  cs : critical section of process i;
    $Q[i] := 0$
od

Figure 5.1: Process i of a system of N processes using Peterson's protocol.

The N processes execute their code in a concurrent way according to the following four atomicity assumptions. Here and in what follows, we denote by φ the formula $\forall k\,(k \neq i \rightarrow Q[k] < J[i]) \vee (S[J[i]] \neq i)$.

(1) The assignments '$Q[i] := 0$' and '$J[i] := 1$' are atomic,

(2) the tests '$\neg J[i] < N$' and '$\neg\varphi$' are atomic,

(3) the sequence of the test '$J[i] < N$' followed by the two assignments '$Q[i] := J[i]$; $S[J[i]] := i$' is atomic, and

(4) the sequence of the command '**await** φ' followed by the assignment '$J[i] := J[i]+1$' is atomic.

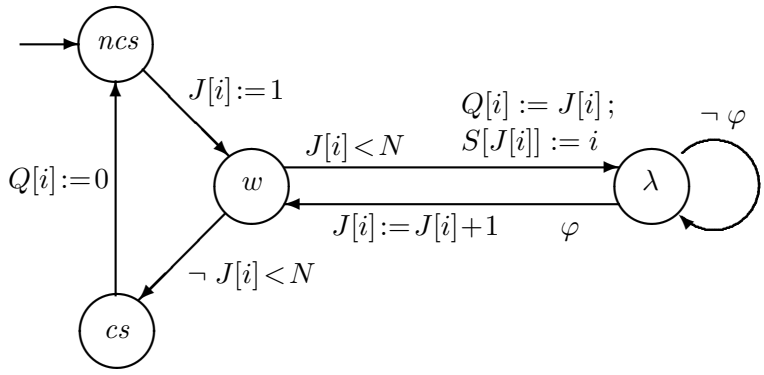

Figure 5.2: Finite state diagram corresponding to process i of a system of N processes using Peterson's protocol. *ncs* is the initial state. The formula φ stands for $\forall k\,(k \neq i \rightarrow Q[k] < J[i]) \vee (S[J[i]] \neq i)$.

We have made these atomicity assumptions (which correspond to the labels of the transitions of the diagram of Figure 5.2) for keeping the presentation of our proof of the mutual exclusion property as simple as possible. However, this property has also been proved by using our method which we will present in Section 5.3, under weaker assumptions, in which one only assumes that every single assignment and test is atomic [74]. (In particular, in [74] it is assumed that each test $k \neq i$ and '$Q[k] < J[i]$' in the formula φ, and *not* the entire formula φ, is atomic. Likewise, it is assumed that in the transition from state w to state λ, each assignment '$Q[i] := J[i]$' and '$S[J[i]] := i$', and *not* the sequence '$Q[i] := J[i]; S[J[i]] := i$' of assignments, is atomic.)

We assume that the number N of processes does *not* change over time, in the sense that while the computation progresses, neither a new process is constructed nor an existing process is destroyed.

In the original paper [51], the proof of the mutual exclusion property of the parameterized Peterson's protocol is left to the reader. The author of [51] simply says that it can be derived from the proof provided for the case of two processes (and, actually, that proof is an informal one) by observing

that, for each value of $J[i] = 1, \ldots, N-1$, at least one process is discarded from the set of those which may enter their critical section. Thus, at the end of the for-loop, at most one process may enter its critical section.

In Peterson's protocol, the value of the variable $J[i]$ of process i indicates, as we will now explain, the 'level' that process i has reached since it first requested to enter its critical section (and this request was done by starting the execution of the while-loop with label w, see Figure 5.1). When process i completes its non-critical section and requests to enter its critical section, it goes to state w where its level $J[i]$ is 1. When process i completes one execution of the body of the while-loop with label w (that is, it goes from state w to state λ and back to state w), it increases its level by one unit. For each level $J[i] = 1, \ldots, N-1$, process i tests whether or not property φ holds, and for $J[i] = 1, \ldots, N-2$, if φ holds at level $J[i]$, then process i goes to the next level up, that is, $J[i]$ is increased by one unit. If φ holds at the final level $N-1$, then process i enters its critical section.

5.2 Specification of Parameterized Protocols Using Array Formulas

In this section we present our method for the specification of parameterized protocols and their temporal properties. The main novelty of our method with respect to other methods based on logic programming is that in the specification of protocols we use the first order theory of arrays introduced below.

Similarly to the model checking approach, we represent a protocol as a set of transitions between states. Notice, however, that in the case of parameterized protocols the number of states may be infinite. For the formal specification of the transition relation we consider a *typed first order language* [43] with the following two types: (i) $\mathbb{N}$, denoting natural numbers, and (ii) $\mathbb{A}$, denoting arrays of natural numbers. A *state* is represented by a term of the form $s(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$ are variables of type $\mathbb{N}$ or $\mathbb{A}$. The *transition relation* is specified by a set of statements of the form:

$t(a, a') \leftarrow \tau$

where t is a fixed binary predicate symbol, a and a' are terms representing states, and τ is an *array formula* defined as we now describe.

An array formula is a typed first order formula constructed by using a language consisting of: (i) *variables* of type $\mathbb{N}$, (ii) *variables* of type $\mathbb{A}$ (called *array variables*), (iii) the constant 0 of type $\mathbb{N}$ and the successor function *succ* of type $\mathbb{N} \rightarrow \mathbb{N}$, and (iv) the following predicates, whose informal meaning is given between parentheses (the names *rd* and *wr* stand for *read* and *write*, respectively):

ln of type $\mathbb{A}\times\mathbb{N}$	($ln(A, l)$ means 'the array A has length l')
rd of type $\mathbb{A}\times\mathbb{N}\times\mathbb{N}$	($rd(A, i, n)$ means 'in the array A the i-th element is n')
wr of type $\mathbb{A}\times\mathbb{N}\times\mathbb{N}\times\mathbb{A}$	($wr(A, i, n, B)$ means 'the array B is equal to the array A except that the i-th element of B is n')
$=_{\mathbb{N}}$, $<$, $\leq$, all of type $\mathbb{N}\times\mathbb{N}$	(equality and inequalities between natural numbers)
$=_{\mathbb{A}}$ of type $\mathbb{A}\times\mathbb{A}$	(equality between arrays)

Given a term n of type $\mathbb{N}$, the term $succ(n)$ will also be written as $n + 1$. For reasons of simplicity, we will write $=$, instead of $=_{\mathbb{N}}$ and $=_{\mathbb{A}}$, when the type of the equality is clear from the context.

Array formulas are constructed as usual in typed first order logic by using the connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$, and $\leftrightarrow$, and the quantifiers $\forall$ and $\exists$. However, for every statement of the form $t(a, a') \leftarrow \tau$ which specifies a transition relation, we assume that every array variable occurring in τ is not quantified within τ itself.

The semantics of a statement of the form $t(a, a') \leftarrow \tau$ is defined in a transformational way by transforming this statement into a stratified set of clauses. This set of clauses is obtained by applying the variant of the Lloyd-Topor transformation for typed first order formulas described in [28], called the *typed Lloyd-Topor transformation*. This transformation works like the Lloyd-Topor transformation for untyped first order formulas [43], except that it adds *type atoms* to the bodies of the transformed clauses so that each variable ranges over the domain specified by the corresponding type

atom. In our case, the transformation adds the type atoms $nat(N)$ and $array(A)$ for each occurrence of a variable N of type $\mathbb{N}$ and a variable A of type $\mathbb{A}$, respectively. The definition of the predicates *nat* and *array* is provided by the following definite clauses:

$$nat(0) \leftarrow \qquad array([\,]) \leftarrow$$
$$nat(N{+}1) \leftarrow nat(N) \qquad array([A|As]) \leftarrow nat(A) \wedge array(As)$$

Note that in these clauses arrays are represented as lists. These four clauses are included in a set, called *Arrays*, of definite clauses that also provide the definitions of the predicates *ln*, *rd*, *wr*, $=_{\mathbb{N}}$, $<$, $\leq$, and $=_{\mathbb{A}}$ of our first order language of arrays. In particular, *Arrays* contains the clauses:

$$ln([\,], 0) \leftarrow$$
$$ln([A|As], L) \leftarrow L{=}N{+}1 \wedge ln(As, N)$$
$$rd([A|As], 1, D) \leftarrow A{=}D$$
$$rd([A|As], L, D) \leftarrow L{=}K{+}1 \wedge rd(As, K, D)$$
$$wr([A|As], 1, D, [B|Bs]) \leftarrow B{=}D \wedge As{=}Bs$$
$$wr([A|As], L, D, [B|Bs]) \leftarrow A{=}B \wedge L{=}K{+}1 \wedge wr(As, K, D, Bs)$$

We omit to list here the usual clauses defining the predicates $=_{\mathbb{N}}$, $<$, $\leq$, and $=_{\mathbb{A}}$.

As an example of application of the typed Lloyd-Topor transformation, let us consider the following statement:

$$t(s(A), s(B)) \leftarrow \exists n\ \forall i\ wr(A, i, n, B)$$

where: (i) A and B are array variables, and (ii) $s(A)$ and $s(B)$ are terms representing states. By applying the typed Lloyd-Topor transformation to this statement, we get the following two clauses:

$$t(s(A), s(B)) \leftarrow array(A) \wedge array(B) \wedge nat(N) \wedge \neg newp(A, N, B)$$
$$newp(A, N, B) \leftarrow array(A) \wedge nat(I) \wedge nat(N) \wedge array(B) \wedge \neg wr(A, I, N, B)$$

Given a statement of the form $H \leftarrow \tau$, where H is an atom and τ is an array formula, we denote by $LT_t(H \leftarrow \tau)$ the set of clauses which are derived by applying the typed Lloyd-Topor transformation to $H \leftarrow \tau$. For reasons of conciseness, in what follows we will feel free to write statements with

array formulas, instead of the corresponding set of clauses, and by abuse of language, statements with array formulas will also be called 'clauses'.

Let us now specify the parameterized Peterson's protocol for N processes by using statements with array formulas. In this specification a *state* is represented by a term of the form $s(P, J, Q, S)$, where:

- P is an array of the form $[p_1, \ldots, p_N]$ such that, for $i = 1, \ldots, N$, p_i is a constant in the set $\{ncs, cs, w, \lambda\}$ representing the state of process i (see Figure 5.2). In order to comply with the syntax of array formulas, the constants ncs, cs, w, and λ should be replaced by distinct natural numbers, but, for reasons of readability, in the formulas below we will use the more expressive identifiers ncs, cs, w, and λ.

- J is an array of the form $[j_1, \ldots, j_N]$, where, for $i = 1, \ldots, N$, j_i belongs to the set $\{1, \ldots, N\}$ and is a local value in the sense that it can be read and written by process i only.

- Q and S are arrays of the form $[q_1, \ldots, q_N]$ and $[s_1, \ldots, s_N]$, respectively, where, for $i = 1, \ldots, N$, q_i belongs to the set $\{0, \ldots, N-1\}$ and s_i belongs to the set $\{1, \ldots, N\}$. These two arrays Q and S are shared in the sense that they can be read and written by any of the N processes.

The transition relation of the parameterized Peterson's protocol is defined by the seven statements $T_1, \ldots, T_7$ which we now introduce. For $r = 1, \ldots, 7$, statement T_r is of the form:

$$t(s(P, J, Q, S), s(P', J', Q', S')) \leftarrow \tau_r(s(P, J, Q, S), s(P', J', Q', S'))$$

where $\tau_r(s(P, J, Q, S), s(P', J', Q', S'))$ is an array formula defined as follows (see also Figure 5.2).

1. For the transition from ncs to w:
$\tau_1(s(P, J, Q, S), s(P', J', Q', S')) \equiv_{def}$
$\exists i \ (rd(P, i, ncs) \ \wedge \ wr(P, i, w, P') \ \wedge \ wr(J, i, 1, J')) \wedge$
$Q' = Q \ \wedge \ S' = S$

2. For the transition from w to λ:
$\tau_2(s(P,J,Q,S), s(P',J',Q',S')) \equiv_{def}$
$\exists i,k,l\ (rd(P,i,w)\ \wedge\ wr(P,i,\lambda,P')\ \wedge\ rd(J,i,k)\ \wedge$
$wr(Q,i,k,Q')\ \wedge\ wr(S,k,i,S')\ \wedge\ ln(P,l)\ \wedge\ k<l\ )\ \wedge$
$J'=J$

3. For the transition from λ to λ:
$\tau_3(s(P,J,Q,S), s(P',J',Q',S')) \equiv_{def}$
$\exists i,k,m,n\ (rd(P,i,\lambda)\ \wedge\ rd(J,i,m)\ \wedge\neg(k=i)\ \wedge\ rd(Q,k,n)\ \wedge$
$n\geq m\ \wedge\ rd(S,m,i))\ \wedge$
$P'=P\ \wedge\ J'=J\ \wedge\ Q'=Q\ \wedge\ S'=S$

4. For the transition from λ to w when $\forall k\,(k\neq i \rightarrow Q[k]<J[i])$ holds:
$\tau_4(s(P,J,Q,S), s(P',J',Q',S')) \equiv_{def}$
$\exists i,l,m\ (rd(P,i,\lambda)\ \wedge\ wr(P,i,w,P')\ \wedge\ rd(J,i,m)\ \wedge\ ln(P,l)\wedge$
$\forall k,n((1\leq k\leq l\ \wedge\ rd(Q,k,n)\ \wedge\neg(k=i))\ \rightarrow\ n<m)\ \wedge$
$wr(J,i,m+1,J'))\ \wedge$
$Q'=Q\ \wedge\ S'=S$

5. For the transition from λ to w when $S[J[i]]\neq i$ holds:
$\tau_5(s(P,J,Q,S), s(P',J',Q',S')) \equiv_{def}$
$\exists i,m\ (rd(P,i,\lambda)\ \wedge\ wr(P,i,w,P')\ \wedge$
$rd(J,i,m)\ \wedge\ \neg\, rd(S,m,i)\ \wedge\ wr(J,i,m+1,J'))\ \wedge$
$Q'=Q\ \wedge\ S'=S$

6. For the transition from w to cs:
$\tau_6(s(P,J,Q,S), s(P',J',Q',S')) \equiv_{def}$
$\exists i,m\ (rd(P,i,w)\ \wedge\ wr(P,i,cs,P')\ \wedge$
$rd(J,i,m)\ \wedge\ ln(P,l)\ \wedge\ m\geq l)\ \wedge$
$J'=J\ \wedge\ Q'=Q\ \wedge\ S'=S$

7. For the transition from cs to ncs:
$\tau_7(s(P,J,Q,S), s(P',J',Q',S')) \equiv_{def}$
$\exists i\ (rd(P,i,cs)\ \wedge\ wr(P,i,ncs,P')\ \wedge\ wr(Q,i,0,Q'))\ \wedge$
$J'=J\ \wedge\ S'=S$

We will express the properties of parameterized protocols by using the branching time temporal logic CTL [18]. In particular, the mutual exclusion property of Peterson's protocol will be expressed by the following temporal formula:

$$\mathit{initial} \rightarrow \neg\, \mathit{EF}\ \mathit{unsafe}$$

where *initial* and *unsafe* are atomic properties of states which we will specify below. This temporal formula holds at a state a whenever the following is true: if a is an *initial* state then there exists no *unsafe* state in the future of a.

The truth of a CTL formula is defined by the following locally stratified logic program, called *Holds*, where the predicate $\mathit{holds}(X, F)$ means that a temporal formula F holds at a state X:

$$\begin{array}{l}
\mathit{holds}(X, F) \leftarrow \mathit{atomic}(X, F) \\
\mathit{holds}(X, \neg\, F) \leftarrow \neg\, \mathit{holds}(X, F) \\
\mathit{holds}(X, F \wedge G) \leftarrow \mathit{holds}(X, F)\ \wedge\ \mathit{holds}(X, G) \\
\mathit{holds}(X, \mathit{ef}(F)) \leftarrow \mathit{holds}(X, F) \\
\mathit{holds}(X, \mathit{ef}(F)) \leftarrow t(X, X') \wedge \mathit{holds}(X', \mathit{ef}(F))
\end{array}$$

Other connectives, such as $\vee$, $\rightarrow$ and $\leftrightarrow$, defined as usual in terms of $\wedge$ and $\neg$, can be used in CTL formulas. The unary constructor *ef* encodes the temporal operator *EF*. Other temporal operators, such as the operator *AF* which is needed for expressing *liveness* properties, can be defined by using locally stratified logic programs [26, 41]. Here, for reasons of simplicity, we have restricted ourselves to the operator *EF* which is the only operator needed for specifying the mutual exclusion property (which is a *safety* property).

The atomic properties of the states are specified by a set of statements of the form:

$$\mathit{atomic}(a, p) \leftarrow \alpha$$

where a is a term representing a state, p is a constant representing an atomic property, and α is an array formula stating that p holds at state a. We assume that the array variables occurring in α are not quantified

within α itself. In particular, the *initial* and *unsafe* atomic properties are defined by the following two statements A_1 and A_2.

$$A_1\colon\ atomic(s(P,J,Q,S), initial) \leftarrow \exists l\,(\forall k\,(1 \leq k \leq l \rightarrow (rd(P,k,ncs)\ \wedge\ rd(Q,k,0))) \wedge ln(P,l)\ \wedge\ ln(J,l)\ \wedge\ ln(Q,l)\ \wedge\ ln(S,l))$$

$$A_2\colon\ atomic(s(P,J,Q,S), unsafe) \leftarrow \exists i,j\,(rd(P,i,cs)\ \wedge\ rd(P,j,cs)\ \wedge \neg(j=i))$$

The premise of A_1, which will also be denoted by $init\_state(s(P,J,Q,S))$, expresses the fact that in an initial state every process is in its non-critical section, Q is an array whose elements are all 0's, and the arrays P, J, Q, and S have the same length. The premise of A_2, which will also be denoted by $unsafe\_state(s(P,J,Q,S))$, expresses the fact that in an unsafe state at least two distinct processes are in their critical section.

Now we formally define when a CTL formula holds for a specification of a parameterized protocol. Let us consider a protocol specification *Spec* consisting of the following set of statements:

$$Spec:\ \{T_1,\ldots,T_m,A_1,\ldots,A_n\}$$

where: (i) $T_1,\ldots,T_m$ are statements that specify a transition relation, and (ii) $A_1,\ldots,A_n$ are statements that specify atomic properties. We denote by P_{Spec} the following set of clauses:

$$P_{Spec}:\ LT_t(T_1)\cup\cdots\cup LT_t(T_m)\cup LT_t(A_1)\cup\cdots\cup LT_t(A_n)\cup Arrays\cup Holds$$

Given a specification *Spec* of a parameterized protocol and a CTL formula φ, we say that

$$\varphi\ holds\ for\ Spec\ \text{ iff }\ M(P_{Spec}) \models \forall X\ holds(X,\varphi)$$

where $M(P_{Spec})$ denotes the perfect model of P_{Spec}. Note that the existence of $M(P_{Spec})$ is guaranteed by the fact that P_{Spec} is locally stratified [3]. In the next section we will prove the mutual exclusion property for the parameterized Peterson's protocol by proving that

$$M(P_{Peterson}) \models \forall X\ holds(X, initial \rightarrow \neg\, ef(unsafe)) \qquad \text{(ME)}$$

where *Peterson* is the specification of the parameterized Peterson's protocol consisting of the set $\{T_1,\ldots,T_7,A_1,A_2\}$ of statements we have listed above.

Note that the above formula (ME) guarantees the mutual exclusion property of the parameterized Peterson's protocol for any number N (≥ 2) of processes. Indeed, in (ME) the variable X ranges over terms of the form $s(P, J, Q, S)$ and the parameter N of Peterson's protocol is the length of the arrays P, J, Q, and S.

5.3 Transformational Verification of Protocols

In this section we describe our method for the verification of CTL properties of parameterized protocols. This method follows the approach based on program transformation which has been proposed in [55]. As an example of application of our method, we prove that the mutual exclusion property holds for the parameterized Peterson's protocol.

Suppose that, given a specification *Spec* of a parameterized protocol and a CTL property φ, we want to prove that φ holds for *Spec*, that is, $M(P_{Spec}) \models \forall X\ holds(X, \varphi)$. We start off by introducing the statement:

$prop \leftarrow \forall X\ holds(X, \varphi)$

where *prop* is a new predicate symbol. By applying the Lloyd-Topor transformation (for untyped formulas) to this statement and by using the equivalence:

$M(P_{Spec}) \models \forall X, F\ (\neg\, holds(X, F) \leftrightarrow holds(X, \neg\, F))$

we get the following two clauses:

1. $prop \leftarrow \neg\, new1$
2. $new1 \leftarrow holds(X, \neg\, \varphi)$

Our verification method consists in showing $M(P_{Spec}) \models \forall X\ holds(X, \varphi)$ by applying unfold/fold transformation rules that preserve the perfect model [26, 73] and deriving from the program $P_{Spec} \cup \{1, 2\}$ a new program T which contains the clause $prop \leftarrow$.

The soundness of our method is a straightforward consequence of the fact that both the Lloyd-Topor transformation and the unfold/fold transformation rules preserve the perfect model, that is, the following holds:

$M(P_{Spec}) \models \forall X\ holds(X, \varphi)$ iff $M(P_{Spec} \cup \{1, 2\}) \models prop$ iff $M(T) \models prop$

Notice that in the case where T contains no clause for *prop*, we conclude that $M(P_{Spec} \cup \{1, 2\}) \not\models prop$ and, thus, $M(P_{Spec}) \models \exists X\ holds(X, \neg\, \varphi)$. Unfortunately, our method is necessarily incomplete due to the undecidability of CTL for parameterized protocols. Indeed, the unfold/fold transformation may not terminate or it may terminate by deriving a program T that contains one or more clauses of the form $prop \leftarrow Body$, where *Body* is not the empty conjunction.

The application of the unfold/fold transformation rules is guided by a transformation strategy which extends the ones presented in [26, 55] to the case of logic programs with array formulas. Now we outline this strategy and then we will see it in action in the verification of the mutual exclusion property of the parameterized Peterson's protocol.

Our transformation strategy is divided into two phases, called Phase A and Phase B, respectively.

In Phase A we compute a specialized definition of $holds(X, \neg\, \varphi)$ as we now describe. Starting from clause 2 above, we perform the following transformation steps: (i) we unfold clause 2, thereby deriving a new set, say *Cls*, of clauses, (ii) we manipulate the array formulas occurring in the clauses of *Cls*, by replacing these formulas by equivalent ones and by removing each clause whose body contains an unsatisfiable formula, (iii) we introduce definitions of new predicates and we fold every instance of $holds(X, F)$. Starting from each definition of a new predicate, we repeatedly perform the above three transformation steps (i), (ii), and (iii). We stop when we are able to fold all instances of $holds(X, F)$ by using predicate definitions already introduced at previous transformation steps.

In Phase B we derive a new program T where as many predicates as possible are defined either by a single fact or by an empty set of clauses, in the hope that *prop* is among such predicates. In order to derive program T we use the unfolding rule and the *clause removal* rule. In particular, we remove all clauses that define *useless* predicates [26]. Recall that: (i) the set U of all useless predicates of a program P is defined as the largest set such that for every predicate p in U and for every clause C that defines p in P, there exists a predicate q in U which occurs positively in the body of C, and (ii) the removal of the clauses that define useless predicates preserves

the perfect model of the program at hand [26].

This two-phase transformation technique has been fruitfully used for proving properties of infinite state systems in [26].

Let us now show how the mutual exclusion property of the parameterized Peterson's protocol can be verified by using our method based on program transformation. The property φ to be verified is $initial \rightarrow \neg\, ef(unsafe)$. Thus, we start from the statement:

$$mutex \leftarrow \forall X\ holds(X, initial \rightarrow \neg\, ef(unsafe))$$

and, by applying the Lloyd-Topor transformation, we get the following two clauses:

1. $mutex \leftarrow \neg\, new1$
2. $new1 \leftarrow holds(X, initial \wedge ef(unsafe))$

Now we apply our transformation strategy starting from $P_{Peterson} \cup \{1, 2\}$, where $P_{Peterson}$ is the program which encodes the specification of the parameterized Peterson's protocol as described in Section 5.2. Let us now show some of the transformation steps performed during Phase A of this strategy. By unfolding clause 2 we get:

3. $new1 \leftarrow init\_state(s(P, J, Q, S)) \ \wedge\ unsafe\_state(s(P, J, Q, S))$
4. $new1 \leftarrow init\_state(s(P, J, Q, S)) \ \wedge\ t(s(P, J, Q, S), s(P', J', Q', S')) \wedge$
   $holds(s(P', J', Q', S'), ef(unsafe))$

Clause 3 is removed because the array formula

$$init\_state(s(P, J, Q, S)) \ \wedge\ unsafe\_state(s(P, J, Q, S))$$

occurring in its body is unsatisfiable (indeed, every process is initially in its non-critical section and, thus, the initial state is not unsafe). In the next section we will discuss the issue of how to mechanize satisfiability tests.

We unfold clause 4 with respect to the atom with predicate t and we get seven new clauses, one for each statement $T_1, \ldots, T_7$ defining the transition relation (see Section 5.2). The clauses derived from $T_2, \ldots, T_7$ are removed because their bodies contain unsatisfiable array formulas. Thus, after these unfolding steps and removal steps, from clause 2 we get the following clause only:

5. $new1 \leftarrow init\_state(s(P,J,Q,S)) \ \wedge \ \tau_1(s(P,J,Q,S), s(P',J',Q',S')) \wedge$
$holds(s(P',J',Q',S'), ef(unsafe))$

where $\tau_1(s(P,J,Q,S), s(P',J',Q',S'))$ is the array formula defined in Section 5.2. Now let us consider the formula $c_1(s(P',J',Q',S'))$ defined as follows:

$c_1(s(P',J',Q',S')) \equiv_{def}$
$\exists P,J,Q,S \ (init\_state(s(P,J,Q,S)) \wedge \tau_1(s(P,J,Q,S), s(P',J',Q',S')))$

By eliminating from it the existentially quantified variables P, J, Q and S, we obtain the following equivalence:

$$
\begin{array}{ll}
c_1(s(P',J',Q',S')) \leftrightarrow & (C) \\
\quad \exists l,i \ (\forall k((1 \leq k \leq l \ \wedge \neg(k=i)) \rightarrow (rd(P',k,ncs) \ \wedge \ rd(Q',k,0))) \wedge & \\
\qquad rd(P',i,w) \ \wedge \ rd(J',i,1) \ \wedge \ rd(Q',i,0) & \\
\qquad ln(P',l) \ \wedge \ ln(J',l) \ \wedge \ ln(Q',l) \ \wedge \ ln(S',l)) &
\end{array}
$$

Now, in order to fold clause 5 w.r.t. the atom $holds(s(P',J',Q',S'), ef(unsafe))$, a suitable condition has to be fulfilled (see the folding rule for constraint logic programs described in [26]). Let us present this condition in the case of programs with array formulas that we consider in this chapter.

Suppose that we are given a clause of the form $H \leftarrow \alpha \wedge holds(X,\psi) \wedge G$ and we want to fold it by using a (suitably renamed) clause of the form $newp(X) \leftarrow \beta \ \wedge \ holds(X,\psi)$. This folding step is allowed only if we have that $M(Arrays) \models \forall(\alpha \rightarrow \beta)$ holds, that is, $\alpha \wedge \neg\beta$ is unsatisfiable in $M(Arrays)$. If this condition is fulfilled, then by folding we obtain the new clause $H \leftarrow \alpha \ \wedge \ newp(X) \ \wedge \ G$.

Now, in order to fold clause 5, we introduce a new predicate definition of the form:

6. $new2(s(P,J,Q,S)) \leftarrow genc_1(s(P,J,Q,S)) \wedge$
$holds(s(P,J,Q,S), ef(unsafe))$

The formula $genc_1(s(P,J,Q,S))$ is a *generalization* of $c_1(s(P,J,Q,S))$, in the sense that the following holds:

$$M(Arrays) \models \forall P, J, Q, S\ (c_1(s(P, J, Q, S)) \rightarrow genc_1(s(P, J, Q, S)))$$

This ensures that the condition for folding is fulfilled.

As usual in program transformation techniques, this generalization step from c_1 to $genc_1$ requires ingenuity. We will not address here the problem of how to mechanize this generalization step and the other generalization steps required in the remaining part of our program derivation. However, some aspects of this crucial generalization issue will be discussed in Section 5.4.

In our verification of the parameterized Peterson's protocol we introduce the following array formula $genc_1(s(P, J, Q, S))$ which holds iff zero or more processes are in state w and the remaining processes are in state ncs:

$$\begin{array}{l} genc_1(s(P, J, Q, S)) \equiv_{def} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (G) \\ \quad \exists l\ (\forall k\ (1 \leq k \leq l\ \rightarrow ((rd(P, k, ncs)\ \wedge\ rd(Q, k, 0)) \vee \\ \qquad\qquad\qquad\qquad\qquad\qquad (rd(P, k, w)\ \wedge\ rd(J, k, 1)\ \wedge\ rd(Q, k, 0)))) \wedge \\ \qquad ln(P, l)\ \wedge\ ln(J, l)\ \wedge\ ln(Q, l)\ \wedge\ ln(S, l)) \end{array}$$

This formula defining $genc_1(s(P, J, Q, S))$ is indeed a generalization of the formula $c_1(s(P, J, Q, S))$, as the reader may check by looking at the above equivalence (C). By folding clause 5 using the newly introduced clause 6 we get:

$$\begin{array}{ll} 5.\text{f} & new1 \leftarrow initstate(s(P, J, Q, S))\ \wedge\ \tau_1(s(P, J, Q, S), s(P', J', Q', S'))\ \wedge \\ & \qquad new2(s(P', J', Q', S')) \end{array}$$

Now, starting from clause 6, we repeat the transformation steps (i), (ii), and (iii) described above, until we are able to fold every instance of $holds(X, F)$ by using a predicate definition introduced at a previous transformation step. By doing so we terminate Phase A and we derive the following program R where:

- $genc_1$ is defined as indicated in (G),
- $genc_2, \ldots, genc_8$ are defined as indicated in the Appendix B,
- $\tau_1, \ldots, \tau_7$ are the array formulas that define the transition relation as indicated in Section 5.2, and

- the arguments a and a' stand for the states $s(P, J, Q, S)$ and $s(P', J', Q', S')$, respectively.

Program R

1. $mutex \leftarrow \neg\, new1$

5.f $new1 \leftarrow initial(a) \wedge \tau_1(a, a') \wedge new2(a')$

7. $new2(a) \leftarrow genc_1(a) \wedge \tau_1(a, a') \wedge new2(a')$
8. $new2(a) \leftarrow genc_1(a) \wedge \tau_2(a, a') \wedge new3(a')$
9. $new2(a) \leftarrow genc_1(a) \wedge \tau_6(a, a') \wedge new7(a')$

10. $new3(a) \leftarrow genc_2(a) \wedge \tau_1(a, a') \wedge new3(a')$
11. $new3(a) \leftarrow genc_2(a) \wedge \tau_2(a, a') \wedge new3(a')$
12. $new3(a) \leftarrow genc_2(a) \wedge \tau_3(a, a') \wedge new3(a')$
13. $new3(a) \leftarrow genc_2(a) \wedge \tau_4(a, a') \wedge new4(a')$
14. $new3(a) \leftarrow genc_2(a) \wedge \tau_5(a, a') \wedge new5(a')$

15. $new4(a) \leftarrow genc_3(a) \wedge \tau_1(a, a') \wedge new4(a')$
16. $new4(a) \leftarrow genc_3(a) \wedge \tau_2(a, a') \wedge new4(a')$
17. $new4(a) \leftarrow genc_3(a) \wedge \tau_2(a, a') \wedge new6(a')$
18. $new4(a) \leftarrow genc_3(a) \wedge \tau_4(a, a') \wedge new4(a')$
19. $new4(a) \leftarrow genc_3(a) \wedge \tau_6(a, a') \wedge new7(a')$

20. $new5(a) \leftarrow genc_4(a) \wedge \tau_1(a, a') \wedge new5(a')$
21. $new5(a) \leftarrow genc_4(a) \wedge \tau_2(a, a') \wedge new5(a')$
22. $new5(a) \leftarrow genc_4(a) \wedge \tau_3(a, a') \wedge new5(a')$
23. $new5(a) \leftarrow genc_4(a) \wedge \tau_4(a, a') \wedge new5(a')$
24. $new5(a) \leftarrow genc_4(a) \wedge \tau_5(a, a') \wedge new5(a')$
25. $new5(a) \leftarrow genc_4(a) \wedge \tau_6(a, a') \wedge new8(a')$

26. $new6(a) \leftarrow genc_5(a) \wedge \tau_1(a, a') \wedge new6(a')$
27. $new6(a) \leftarrow genc_5(a) \wedge \tau_2(a, a') \wedge new6(a')$
28. $new6(a) \leftarrow genc_5(a) \wedge \tau_3(a, a') \wedge new6(a')$
29. $new6(a) \leftarrow genc_5(a) \wedge \tau_4(a, a') \wedge new6(a')$
30. $new6(a) \leftarrow genc_5(a) \wedge \tau_5(a, a') \wedge new6(a')$
31. $new6(a) \leftarrow genc_5(a) \wedge \tau_6(a, a') \wedge new9(a')$

32. $new7(a) \leftarrow genc_6(a) \wedge \tau_1(a, a') \wedge new7(a')$

33. $new7(a) \leftarrow genc_6(a) \wedge \tau_2(a, a') \wedge new9(a')$
34. $new7(a) \leftarrow genc_6(a) \wedge \tau_7(a, a') \wedge new2(a')$
35. $new8(a) \leftarrow genc_7(a) \wedge \tau_3(a, a') \wedge new8(a')$
36. $new8(a) \leftarrow genc_7(a) \wedge \tau_7(a, a') \wedge new5(a')$
37. $new9(a) \leftarrow genc_8(a) \wedge \tau_1(a, a') \wedge new9(a')$
38. $new9(a) \leftarrow genc_8(a) \wedge \tau_2(a, a') \wedge new9(a')$
39. $new9(a) \leftarrow genc_8(a) \wedge \tau_3(a, a') \wedge new9(a')$
40. $new9(a) \leftarrow genc_8(a) \wedge \tau_5(a, a') \wedge new9(a')$
41. $new9(a) \leftarrow genc_8(a) \wedge \tau_7(a, a') \wedge new6(a')$

Now we proceed to Phase B of our strategy. Since in program R the predicates $new1$ through $new9$ are useless, we remove clause 5.f, and clauses 7 through 41, and by doing so, we derive a program consisting of clause 1 only. By unfolding clause 1 we get the final program T, which consists of the clause $mutex \leftarrow$ only. Thus, $M(T) \models mutex$ and we have proved that:

$$M(P_{Peterson}) \models \forall X \; holds(X, initial \rightarrow \neg \, ef(unsafe))$$

As a consequence, we have that for any initial state and for any number $N(\geq 2)$ of processes, the mutual exclusion property holds for the parameterized Peterson's protocol.

5.4 Mechanization of the Verification Method

In order to achieve a full mechanization of our verification method, two main issues have to be addressed: (i) how to test the satisfiability of array formulas, and (ii) how to perform suitable generalization steps.

Satisfiability tests for array formulas are required at the following two points of Phase A of the transformation strategy described in Section 5.3: (1) at Step (ii), when we remove each clause whose body contains an unsatisfiable array formula, and (2) at Step (iii), when we fold each clause whose body contains a *holds* literal.

In order to clarify Point (2), we recall that, before applying the folding rule [26], we need to test that in $M(Arrays)$ the array formula occurring in the body of the clause to be folded implies the array formula occurring in

the body of the clause that we use for folding. For instance, in Section 5.3 before folding clause 5 using clause 6, we need to prove that:

$$M(Arrays) \models \forall P, J, Q, S\ (c_1(s(P, J, Q, S)) \rightarrow genc_1(s(P, J, Q, S)))$$

which holds iff the following formula:

$$c_1(s(P, J, Q, S)) \wedge \neg\, genc_1(s(P, J, Q, S)) \qquad (CG)$$

is unsatisfiable in $M(Arrays)$.

Now the problem of testing the satisfiability of array formulas is in general undecidable. (The reader may refer to [78] for a short survey on this subject.) However, some decidable fragments of the theory of arrays, such as the *quantifier-free extensional theory of arrays*, have been identified [78]. Unfortunately, the array formulas occurring in our formalization of the parameterized Peterson's protocol cannot be reduced to formulas in those decidable fragments. Indeed, due to the assumptions made in Section 5.2 on the array formulas which are used in the specifications of protocols, we need to test the satisfiability of array formulas where the variables of type $\mathbb{A}$ are not quantified, while the variables of type $\mathbb{N}$ can be quantified in an unrestricted way.

In order to perform the satisfiability tests required by our verification of the parameterized Peterson's protocol, we have followed the approach based on program transformation which has been proposed in [55]. Some of these satisfiability tests have been done in a fully automatic way by using the MAP transformation system [46], which implements the unfold/fold proof strategy described in [55]. Examples of array formulas whose unsatisfiability we have proved in an automatic way include: (i) the formula occurring in the body of clause 3 shown in Section 5.3, and (ii) the formula (CG) shown above in this section. Some other satisfiability tests have been done in a semi-automatic way, by interleaving fully automatic applications of the unfold/fold proof strategy and some manual applications of the unfold/fold transformation rules.

Generalization steps are needed when, during Step (iii) of Phase A of our transformation strategy, a new predicate definition is introduced to fold the instances of the atom $holds(X, F)$. The introduction of suitable new definitions by generalization is a crucial issue of the program transforma-

tion methodology [17]. The invention of these definitions corresponds to the discovery of suitable invariants of the protocol to be verified. Due to the undecidability of CTL for parameterized protocols, it is impossible to provide a general, fully automatic technique which performs the suitable generalization steps in all cases. However, we have followed an approach that, in the case of the parameterized Peterson's protocol, works in a systematic way. This approach extends to the case of logic programs with array formulas some generalization techniques which are used for the specialization of (constraint) logic programs [25, 40] and it can be briefly described as follows.

The new predicate definitions introduced during Step (iii) of Phase A of the transformation strategy are arranged as a tree *DefsTree* of clauses. The root of *DefsTree* is clause 2. Given a clause N, the children of N are the predicate definitions which are introduced to fold the instances of $holds(X, F)$ in the bodies of the clauses obtained by unfolding N at Step (i) and not removed at Step (ii).

If the new predicate definitions are introduced without any guidance, then we may construct a tree *DefsTree* with infinite paths, and the transformation strategy may not terminate. In order to avoid the construction of infinite paths and achieve the termination of the transformation strategy, before adding a new predicate definition D to *DefsTree* as a child of a clause N, we match D against every clause A occurring in the path from the root of *DefsTree* to N. Suppose that A is of the form $newp(X) \leftarrow \alpha \wedge holds(X, \psi)$ and D is of the form $newq(X) \leftarrow \delta \wedge holds(X, \psi)$. If the array formula α is *embedded* (with respect to a suitable ordering) into the array formula δ, then instead of introducing D, we introduce a clause of the form $gen(X) \leftarrow \gamma \wedge holds(X, \psi)$, where γ is a generalization of both α and δ, that is, both $M(Arrays) \models \forall\, (\alpha \rightarrow \gamma)$ and $M(Arrays) \models \forall\, (\delta \rightarrow \gamma)$ holds.

Thus, in order to fully mechanize our generalization technique we need to address the following two problems: (i) the introduction of a formal definition of the *embedding* relation between array formulas, and (ii) the computation of the array formula γ from α and δ. Providing solutions to these two problems is beyond the scope of the present chapter. However, a possible approach to follow for solving these problems consists in extending

to logic programs with array formulas the notions that have been introduced in the area of specialization of (constraint) logic programs (see, for instance, [25, 40]).

5.5 Related Work and Conclusions

The method for protocol verification presented in this chapter is based on the program transformation approach which has been proposed in [55] for the verification of properties of locally stratified logic programs. We consider concurrent systems of *finite state* processes. We assume that systems are *parameterized*, in the sense that they consist of an *arbitrary* number of processes. We also assume that parameterized systems may use arrays of parameterized length. The properties of the systems we want to verify, are the temporal logic properties which can be expressed in CTL (Computational Tree Logic) [18]. Our method consists in: (i) encoding a parameterized system and the property to be verified as a locally stratified logic program extended with array formulas, and then (ii) applying suitable unfold/fold transformations to this program so to derive a new program where it is immediate to check whether or not the property holds.

In general, the problem of verifying CTL properties of parameterized systems is undecidable [4] and thus, in order to find decision procedures, one has to consider subclasses of systems where the problem is decidable. Some of these decidable subclasses in the presence of arrays have been studied in [39], but unfortunately, our formalization of the parameterized Peterson's protocol does not belong to any of those classes, because it requires more than two arrays of natural numbers, and also requires assignments and reset operations.

As yet, our method is *not* fully mechanical and human intervention is needed for: (i) the test of satisfiability for array formulas, and (ii) the introduction of new definitions by generalization. We have discussed these two issues in Section 5.4.

Other verification methods for concurrent systems based on the transformational approach are those presented in [26, 28, 41, 69, 68].

In [26] it is presented a method for verifying CTL properties of systems

consisting of a *fixed number* of infinite state processes. That method makes use of locally stratified constraint logic programs, where the constraints are linear equations and disequations on real numbers. In this chapter we have followed an approach similar to constraint logic programming, but in our setting we have array formulas, instead of constraints. The method presented here can easily be extended to deal with parameterized infinite state systems by considering, for instance, arrays of infinite state processes.

The paper [28] describes the verification of the mutual exclusion property for the parameterized Bakery protocol which was introduced in [37]. In [28] the authors use locally stratified logic programs extended with formulas of the Weak Monadic Second Order Theory of k-Successors (WSkS) which expresses monadic properties of strings. The array formulas considered in this chapter are more expressive than WSkS formulas, because array formulas can express polyadic properties. However, there is price to pay, because in general the theory of array formulas is undecidable, while the theory WSkS is decidable.

The method described in [41] uses partial deduction and abstract interpretation of logic programs for verifying safety properties of infinite state systems. Partial deduction is strictly less powerful than unfold/fold program transformation, which, on the other hand, is more difficult to mechanize when unrestricted transformations are considered. One of the main objectives of our future research is the design of suitably restricted unfold/fold transformations which are easily mechanizable and yet powerful enough for the verification of program properties.

The work presented in [69, 68] is the most similar to ours. The authors of these two papers use unfold/fold rules for transforming programs and proving properties of parameterized concurrent systems. Our results differs from those presented in [69, 68] in that, instead of using definite logic programs, we use logic programs with locally stratified negation and array formulas for the specification of concurrent systems and their properties. As a consequence, also the transformation rules we consider are different and more general than those used in [69, 68].

Besides the above mentioned transformational methods, some more verification methods based on (constraint) logic programming have been pro-

posed in the literature [22, 30, 50, 63].

The methods proposed in [50, 63] deal with finite state systems only. In particular, the method presented in [50] uses constraint logic programming with finite domains, extended with constructive negation and tabled resolution, for finite state local model checking, and the method described in [63] uses tabled logic programming to efficiently verify μ-calculus properties of finite state systems expressed in the CCS calculus.

The methods presented in [22, 30] deal with infinite state systems. In particular, [22] describes a method which is based on constraint logic programming and can be applied for verifying CTL properties of infinite state systems by computing approximations of least and greatest fixpoints via abstract interpretation. An extension of this method has also been used for the verification of parameterized cache coherence protocols [21]. The method described in [30] uses logic programs with linear arithmetic constraints and Presburger arithmetic for the verification of safety properties of Petri nets. Unfortunately, however, parameterized systems that use arrays, like Peterson's protocol, cannot be directly specified and verified using the methods considered in [22, 30] because, in general, array formulas cannot be encoded as constraints over real numbers or Presburger formulas.

Several other verification techniques for parameterized systems have been proposed in the literature outside the area of logic programming (see [83] for a survey of some of these techniques). These techniques extend finite state model checking with various forms of *abstraction* (for reducing the verification of a parameterized system to the verification of a finite state system) or *induction* (for proving properties for every value of the parameter).

We do not have space here to discuss the relationships of our work with the many techniques for proving properties based on abstraction. We only want to mention the technique proposed in [5], which has also been applied for the verification of the parameterized Peterson's protocol. That technique can be applied for verifying in an automatic way safety properties of all systems that satisfy a so-called *stratification* condition. Indeed, when this condition holds for a given parameterized system, then the verification task can be reduced to the verification of a finite number of finite state

systems that are instances of the given parameterized system for suitable values of the parameter. However, Peterson's protocol does *not* satisfy the stratification condition and its treatment with the technique proposed in [5] requires a significant amount of ingenuity.

Our verification method is also related to the verification techniques based on induction (see, for instance, [48]). These techniques use interactive theorem proving tools where many tasks are mechanized, but the construction of a whole proof requires substantial human guidance. Our method has advantages and disadvantages with respect to these techniques based on induction. On one hand, in our approach we need neither explicit induction on the parameter of the system nor the introduction of suitable induction hypotheses. On the other hand, as already mentioned, our method needs suitable generalization steps which cannot be fully mechanized.

Chapter 6

Appendix A

In this chapter we present some example transformations that we have run by using our implementation, in the MAP Transformation System, of the method presented in Chapter 2. The MAP transformation system runs on SICStus Prolog 3.12.5 and is available at:

`http://www.iasi.cnr.it/~proietti/system.html`

A.1 In Correct Position

The following example is taken from [31] and consists in deriving a recursive definition for the predicate *In_Correct_Position*. At the end of the transformation, we prove that the transformation is totally correct by computing positive weights for the clauses of the final program such that the associated Correctness Constraint System is satisfied.

Initial program:

```
1. icp(A,B) :- inodd(A,B), odd(A).
2. icp(A,B) :- ineven(A,B), even(A).
3. inodd(A,[A|B]).
4. inodd(A,[B,C|D]) :- inodd(A,D).
5. ineven(A,[B,A|C]).
6. ineven(A,[B,C|D]) :- ineven(A,D).
```

```
7. odd(s(A)) :- even(A).
8. even(0).
9. even(s(A)) :- even(A).

| ?- u.
clause number  [no]: 1
Number  [1]:
by unfolding clause 1 w.r.t. atom 1 we get:
10. icp(A,[A|B]) :- odd(A).
11. icp(A,[B,C|D]) :- inodd(A,D), odd(A).

| ?- u.
clause number  [no]: 2
Number  [1]:
by unfolding clause 2 w.r.t. atom 1 we get:
12. icp(A,[B,A|C]) :- even(A).
13. icp(A,[B,C|D]) :- ineven(A,D), even(A).

| ?- f([11,13],[1,2]).
by folding clause [11,13] using [1,2] we get:
14. icp(A,[B,C|D]) :- icp(A,D).

| ?- pp.
Current program:

10. icp(A,[A|B]) :- odd(A).
14. icp(A,[B,C|D]) :- icp(A,D).
12. icp(A,[B,A|C]) :- even(A).

| ?- current_constraint_system.

Clause weights:

[(10,w(10)),(14,w(14)),(12,w(12)),(3,w(3)),(4,w(4)),(5,w(5)),
 (6,w(6)),(7,w(7)),(8,w(8)),(9,w(9))]

The Correctness Constraint System of the Current Program is:

[w(10)=:=w(1)+w(3),w(11)=:=w(1)+w(4),w(12)=:=w(2)+w(5),
 w(13)=:=w(2)+w(6),w(14)=<w(11)-w(1),w(14)=<w(13)-w(2)]
```

```
| ?- solve_constraint_system.
The Correctness Constraint System of the Current Program is SATISFIABLE.

A solution is:
Clause number 10 weight 1
Clause number 14 weight 1
Clause number 12 weight 1
Clause number 3 weight 1
Clause number 4 weight 1
Clause number 5 weight 1
Clause number 6 weight 1
Clause number 7 weight 1
Clause number 8 weight 1
Clause number 9 weight 1

Elapsed time: 10ms
```

A.2 Adjacent

The following example is taken from [33] and consists in deriving a more efficient definition for the predicate *Adjacent* which checks whether or not two elements have adjacent occurrences in a list. At the end of the transformation, we prove that the transformation is totally correct by computing positive weights for the clauses of the final program such that the associated Correctness Constraint System is satisfied.

```
Initial program:

1. a([],A,A).
2. a([A|B],C,[A|D]) :- a(B,C,D).

| ?- d.
Do you want to check the existing definitions [n]?
enter clauses (use the same new predicate in the heads):
first clause: adj(A,B,C) :- a(D,[A,B],E), a(E,F,C).
{Warning: [D] - singleton variable.}
{Warning: [F] - singleton variable.}
Another clause [n]?
```

```
3. adj(A,B,C) :- a(D,[A,B],E), a(E,F,C).

| ?- u.
clause number  [no]: 3
Number  [1]: 1
by unfolding clause 3 w.r.t. atom 1 we get:
4. adj(A,B,C) :- a([A,B],D,C).
5. adj(A,B,C) :- a(D,[A,B],E), a([F|E],G,C).

| ?- u.
clause number  [no]: 4
Number  [1]:
by unfolding clause 4 w.r.t. atom 1 we get:
6. adj(A,B,[A|C]) :- a([B],D,C).

| ?- u.
clause number  [no]: 6
Number  [1]:
by unfolding clause 6 w.r.t. atom 1 we get:
7. adj(A,B,[A,B|C]) :- a([],D,C).

| ?- u.
clause number  [no]: 7
Number  [1]:
by unfolding clause 7 w.r.t. atom 1 we get:
8. adj(A,B,[A,B|C]).

| ?- u.
clause number  [no]: 5
Number  [1]: 2
by unfolding clause 5 w.r.t. atom 2 we get:
9. adj(A,B,[C|D]) :- a(E,[A,B],F), a(F,G,D).

| ?- f([9],[3]).
by folding clause [9] using [3] we get:
10. adj(A,B,[C|D]) :- adj(A,B,D).

| ?- current_constraint_system.

Clause weights:
[(1,w(1)),(2,w(2)),(8,w(8)),(10,w(10))]
```

```
The Correctness Constraint System for the Current Program is:

[w(4)=:=w(3)+w(1),w(5)=:=w(3)+w(2),w(6)=:=w(4)+w(2),w(7)=:=w(6)+w(2),
 w(8)=:=w(7)+w(1),w(9)=:=w(5)+w(2),w(10)=<w(9)-w(3)]

| ?- solve_constraint_system.
The Correctness Constraint System for the Current Program is SATISFIABLE.

A solution is:
Clause number 1 weight 1
Clause number 2 weight 1
Clause number 8 weight 4
Clause number 10 weight 1

Elapsed time 10ms
```

A.3 Proving a Liveness Property of an N-bit Shift Register

The following example is taken from [66]. The transformation consists in proving a liveness property of an n-bit shift register whose specification is given as a logic program. In particular, we first apply the Unfold/Fold proof method illustrated in Chapter 2 and prove the equivalence of the two goals `test(A)` and `test1(A)`. Then, we use this equivalence to apply a goal replacement rule during the proof of the liveness property.

```
Initial program:

1. thm(A) :- gen(A), test(A).
2. gen([]).
3. gen([0|A]) :- gen(A).
4. test(A) :- canon(A).
5. test(A) :- trans(A,B), test(B).
6. canon([]).
7. canon([1|A]) :- canon(A).
8. trans([0|A],[1|A]).
9. trans([1|A],[1|B]) :- trans(A,B).
```

```
10. test1(A) :- test([1|A]).
11. new1(A) :- test(A).
12. new2(A) :- test1(A).

| ?- u.
clause number  [no]: 11
Number  [1]:
by unfolding clause 11 w.r.t. atom 1 we get:
13. new1(A) :- canon(A).
14. new1(A) :- trans(A,B), test(B).

| ?- s_f([14],[11]).
by symmetric folding clause [14] using [11] we get:
15. new1(A) :- trans(A,B), new1(B).

| ?- u.
clause number  [no]: 12
Number  [1]:
by unfolding clause 12 w.r.t. atom 1 we get:
16. new2(A) :- test([1|A]).

| ?- u.
clause number  [no]: 16
Number  [1]:
by unfolding clause 16 w.r.t. atom 1 we get:
17. new2(A) :- canon([1|A]).
18. new2(A) :- trans([1|A],B), test(B).

| ?- u.
clause number  [no]: 17
Number  [1]:
by unfolding clause 17 w.r.t. atom 1 we get:
19. new2(A) :- canon(A).

| ?- u.
clause number  [no]: 18
Number  [1]:
by unfolding clause 18 w.r.t. atom 1 we get:
20. new2(A) :- trans(A,B), test([1|B]).

| ?- f([20],[10]).
```

```
by folding clause [20] using [10] we get:
21. new2(A) :- trans(A,B), test1(B).

| ?- f([21],[12]).
by folding clause [21] using [12] we get:
22. new2(A) :- trans(A,B), new2(B).

| ?- pp.
Current program:
13. new1(A) :- canon(A).
15. new1(A) :- trans(A,B), new1(B).
19. new2(A) :- canon(A).
22. new2(A) :- trans(A,B), new2(B).

| ?- uf_proof(new2/1,new1/1).

The transformation done for new1 is symmetrical.

The predicates new2/1 and new1/1 are syntactically equivalent
under the renaming: [new2/new1]

The Correctness Constraint System of the Current Program is SATISFIABLE.

A solution is:
Clause number w(1) weight 1
Clause number w(2) weight 1
Clause number w(3) weight 1
Clause number w(4) weight 1
Clause number w(5) weight 1
Clause number w(6) weight 1
Clause number w(7) weight 1
Clause number w(8) weight 1
Clause number w(9) weight 1
Clause number w(10) weight 1
Clause number w(11) weight 1
Clause number w(12) weight 1
Elapsed time 10ms

The following Law

test1(A) -> test(A)
```

```
can be applied under the following constraint

Clause weights :

[w(1),w(2),w(3),w(4),w(5),w(6),w(7),w(8),w(9),w(10),w(11),w(12)]

Constraint:

[w(7)+w(10)-w(11)+w(12)>=0,w(9)>=0]

Law have been asserted with ID 1.

Do you want to add to the program file, the computed law [n]? y
Law successfully saved in current program file.
```

The Law 1, that we have just proved to hold in the model of the initial program, is going to be used in the following transformation. In particular, the application of the goal replacement rule will add to the current Correctness Constraint System the constraints computed in the previous transformation, which ensure the correctness of the goal replacement.

```
Initial program:

1. thm(A) :- gen(A), test(A).
2. gen([]).
3. gen([0|A]) :- gen(A).
4. test(A) :- canon(A).
5. test(A) :- trans(A,B), test(B).
6. canon([]).
7. canon([1|A]) :- canon(A).
8. trans([0|A],[1|A]).
9. trans([1|A],[1|B]) :- trans(A,B).
10. test1(A) :- test([1|A]).
11. new1(A) :- test(A).
12. new2(A) :- test1(A).

| ?- laws.

1. test1(A) -> test(A).
```

```
| ?- u.
clause number  [no]: 1
Number  [1]:
by unfolding clause 1 w.r.t. atom 1 we get:
13. thm([]) :- test([]).
14. thm([0|A]) :- gen(A), test([0|A]).

| ?- u.
clause number  [no]: 13
Number  [1]:
by unfolding clause 13 w.r.t. atom 1 we get:
15. thm([]) :- canon([]).
16. thm([]) :- trans([],A), test(A).

| ?- u.
clause number  [no]: 15
Number  [1]:
by unfolding clause 15 w.r.t. atom 1 we get:
17. thm([]).

| ?- del_f_cl.
deletion of clause 16 because of failure in body

| ?- pp.
17. thm([]).
14. thm([0|A]) :- gen(A), test([0|A]).

| ?- u.
clause number  [no]: 14
Number  [1]: 2
by unfolding clause 14 w.r.t. atom 2 we get:
18. thm([0|A]) :- gen(A), canon([0|A]).
19. thm([0|A]) :- gen(A), trans([0|A],B), test(B).

| ?- del_f_cl.
deletion of clause 18 because of failure in body

| ?- u.
clause number  [no]: 19
Number  [1]: 2
```

```
by unfolding clause 19 w.r.t. atom 2 we get:
20. thm([0|A]) :- gen(A), test([1|A]).

| ?- u.
clause number  [no]: 20
Number  [1]: 2
by unfolding clause 20 w.r.t. atom 2 we get:
21. thm([0|A]) :- gen(A), canon([1|A]).
22. thm([0|A]) :- gen(A), trans([1|A],B), test(B).

| ?- u.
clause number  [no]: 21
Number  [1]: 2
by unfolding clause 21 w.r.t. atom 2 we get:
23. thm([0|A]) :- gen(A), canon(A).

| ?- u.
clause number  [no]: 22
Number  [1]: 2
by unfolding clause 22 w.r.t. atom 2 we get:
24. thm([0|A]) :- gen(A), trans(A,B), test([1|B]).

| ?- f([24],[10]).
by folding clause [24] using [10] we get:
25. thm([0|A]) :- gen(A), trans(A,B), test1(B).

| ?- rep.
clause number  [no]: 25
atom numbers [no]: 3
law number  [no]: 1

by replacing goal [test1(A)]
by [test(A)]
in clause 25 using law 1 we get:
26. thm([0|A]) :- gen(A), trans(A,B), test(B).

| ?- f([23,26],[4,5]).
by folding clause [23,26] using [4,5] we get:
27. thm([0|A]) :- gen(A), test(A).

| ?- f([27],[1]).
```

```
by folding clause [27] using [1] we get:
28. thm([0|A]) :- thm(A).

| ?- current_constraint_system.

Clause weights:

[(17,w(17)),(28,w(28)),(2,w(2)),(3,w(3)),(4,w(4)),(5,w(5)),(6,w(6)),
 (7,w(7)),(8,w(8)),(9,w(9)),(10,w(10)),(11,w(11)),(12,w(12))]

The Correctness Constraint System for the Current Program is:

[w(13)=:=w(1)+w(2),w(14)=:=w(1)+w(3),w(15)=:=w(13)+w(4),
 w(16)=:=w(13)+w(5),w(17)=:=w(15)+w(6),w(18)=:=w(14)+w(4),
 w(19)=:=w(14)+w(5),w(20)=:=w(19)+w(8),w(21)=:=w(20)+w(4),
 w(22)=:=w(20)+w(5),w(23)=:=w(21)+w(7),w(24)=:=w(22)+w(9),
 w(25)=<w(24)-w(10),w(26)=:=w(25)+(w(11)-w(12)),w(7)+w(10)-w(11)+w(12)>=0,
 w(9)>=0,w(27)=<w(23)-w(4),w(27)=<w(26)-w(5),w(28)=<w(27)-w(1)]

| ?- solve_constraint_system.
The Correctness Constraint System for the Current Program is SATISFIABLE.

A solution is:
Clause number 17 weight 3
Clause number 28 weight 1
Clause number 2 weight 1
Clause number 3 weight 1
Clause number 4 weight 1
Clause number 5 weight 1
Clause number 6 weight 1
Clause number 7 weight 1
Clause number 8 weight 1
Clause number 9 weight 1
Clause number 10 weight 1
Clause number 11 weight 1
Clause number 12 weight 1

Elapsed time 10ms
```

A.3 Equal Frontiers

The following example is taken from [17, 80] and consists in transforming the initial program *Equal Frontiers*, which checks whether or not the frontiers of two binary trees are equal, into a computationally more efficient program. The following is the initial program and the goal replacement laws 1,...,5 that have been proved correct by using out Unfold/Fold proof method.

```
P1: Progs/ts_frontiers.pl
1. frontier(tree(A,B),C) :- frontier(A,D), frontier(B,E), append(D,E,C).
2. frontier(tip(A),[A]).
3. eqtree(A,B) :- frontier(A,C), frontier(B,C).
4. frontier1(A,B) :- frontierlist(A,C), flatten(C,B).
5. frontierlist([],[]).
6. frontierlist([A|B],[C|D]) :- frontier(A,C), frontierlist(B,D).
7. flatten([],[]).
8. flatten([A|B],C) :- flatten(B,D), append(A,D,C).
9. eqtreelist(A,B) :- frontier1(A,C), frontier1(B,C).
10. append([],A,A).
11. append([A|B],C,[A|D]) :- append(B,C,D).
12. list([]).
13. list([A|B]) :- list(B).

| ?- laws.
1. append(A,B,C), append(C,D,E) -> append(B,D,F), append(A,F,E).
2. frontier(A,B) -> frontier(A,C), append(C,[],B).
3. frontierlist([A],[B]), flatten([B],C) ->
   frontierlist([A],D), flatten(D,C).
4. flatten([],[]), append(A,[],B) -> flatten([],C), append(A,C,B).
5. frontierlist([A,B|C],[D,E|F]), flatten([D,E|F],G) ->
   frontierlist([A,B|C],H), flatten(H,G).
```

For reasons of conciseness, the following transformation is provided in the form of a transformation history, where useless details are omitted. In particular, by using the notation `Pn: by ... Pm ...` we mean that the program `Pn` has been obtained by application of a transformation rule to a program `Pm`, where `m<n`. Also, at the end of a line `Pm: by ...` we sometimes add the string `(...Pn)`, with `m<n`, to denote that the program `Pn` has been

obtained form the program `Pm` by using `m-n` goal rearrangement steps and that we omit to indicate programs `Pm`, ..., `Pn-1`.

```
P2: by replacing atoms [1] in cl 3 in P1 using law 2
24. eqtree(A,B) :- frontier(A,C), append(C,[],D), frontier(B,D).

P3: by replacing atoms [3] in cl 24 in P2 using law 2
25. eqtree(A,B) :- frontier(A,C), append(C,[],D), frontier(B,E),
    append(E,[],D).

P4: by folding cl 25 in P3 using cl 7
26. eqtree(A,B) :- frontier(A,C), append(C,[],D), frontier(B,E),
    append(E,[],D), flatten([],[]).

P5: by folding cl 26 in P4 using cl 7
27. eqtree(A,B) :- frontier(A,C), append(C,[],D), frontier(B,E),
    append(E,[],D), flatten([],[]), flatten([],[]).

P6: by replacing atoms [5,2] in cl 27 in P5 using law 4
28. eqtree(A,B) :- frontier(A,C), frontier(B,D), append(D,[],E),
    flatten([],[]), flatten([],F), append(C,F,E).

P7: by replacing atoms [4,3] in cl 28 in P6 using law 4
29. eqtree(A,B) :- frontier(A,C), frontier(B,D), flatten([],E),
    flatten([],F), append(D,F,G), append(C,E,G).

P8: by folding cl 29 in P7 using cl 8 (...P9)
31. eqtree(A,B) :- frontier(A,C), frontier(B,D), flatten([],E),
    append(C,E,F), flatten([D],F).

P10: by folding cl 31 in P9 using cl 8
32. eqtree(A,B) :- frontier(A,C), frontier(B,D), flatten([C],E),
    flatten([D],E).

P11: by folding cl 32 in P10 using cl 5
33. eqtree(A,B) :- frontier(A,C), frontier(B,D), flatten([C],E),
    flatten([D],E), frontierlist([],[]).

P12: by folding cl 33 in P11 using cl 5 (...P14)
36. eqtree(A,B) :- frontier(A,C), frontierlist([],[]), frontier(B,D),
    frontierlist([],[]), flatten([C],E), flatten([D],E).
```

```
P15: by folding cl 36 in P14 using cl 6
37. eqtree(A,B) :- frontierlist([A],[C]), frontier(B,D), frontierlist([],[]),
    flatten([C],E), flatten([D],E).

P16: by folding cl 37 in P15 using cl 6
38. eqtree(A,B) :- frontierlist([A],[C]), frontierlist([B],[D]),
    flatten([C],E), flatten([D],E).

P17: by replacing atoms [1,3] in cl 38 in P16 using law 3
39. eqtree(A,B) :- frontierlist([A],C), flatten(C,D), frontierlist([B],[E]),
    flatten([E],D).

P18: by replacing atoms [3,4] in cl 39 in P17 using law 3
40. eqtree(A,B) :- frontierlist([A],C), flatten(C,D), frontierlist([B],E),
    flatten(E,D).

P19: by folding cl 40 in P18 using cl 4
41. eqtree(A,B) :- frontier1([A],C), frontierlist([B],D), flatten(D,C).

P20: by folding cl 41 in P19 using cl 4
42. eqtree(A,B) :- frontier1([A],C), frontier1([B],C).

P21: by folding cl 42 in P20 using cl 9
43. eqtree(A,B) :- eqtreelist([A],[B]).

P22: by unfolding cl 4 in P21 wrt atom 1 using P1
43. eqtree(A,B) :- eqtreelist([A],[B]).
44. frontier1([],A) :- flatten([],A).
45. frontier1([A|B],C) :- frontier(A,D), frontierlist(B,E), flatten([D|E],C).

P23: by unfolding cl 44 in P22 wrt atom 1 using P1
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
45. frontier1([A|B],C) :- frontier(A,D), frontierlist(B,E), flatten([D|E],C).

P24: by unfolding cl 45 in P23 wrt atom 3 using P1
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
47. frontier1([A|B],C) :- frontier(A,D), frontierlist(B,E), flatten(E,F),
    append(D,F,C).
```

```
P25: by unfolding cl 47 in P24 wrt atom 1 using P1
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
48. frontier1([tree(A,B)|C],D) :- frontier(A,E), frontier(B,F), append(E,F,G),
    frontierlist(C,H), flatten(H,I), append(G,I,D).
49. frontier1([tip(A)|B],C) :- frontierlist(B,D), flatten(D,E), append([A],E,C).

P26: by unfolding cl 49 in P25 wrt atom 3 using P1
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
48. frontier1([tree(A,B)|C],D) :- frontier(A,E), frontier(B,F), append(E,F,G),
    frontierlist(C,H), flatten(H,I), append(G,I,D).
50. frontier1([tip(A)|B],[A|C]) :- frontierlist(B,D), flatten(D,E),
    append([],E,C).

P27: by unfolding cl 50 in P26 wrt atom 3 using P1
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
48. frontier1([tree(A,B)|C],D) :- frontier(A,E), frontier(B,F), append(E,F,G),
    frontierlist(C,H), flatten(H,I), append(G,I,D).
51. frontier1([tip(A)|B],[A|C]) :- frontierlist(B,D), flatten(D,C).

P28: by folding cl 51 in P27 using cl 4
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
48. frontier1([tree(A,B)|C],D) :- frontier(A,E), frontier(B,F), append(E,F,G),
    frontierlist(C,H), flatten(H,I), append(G,I,D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).

P29: by replacing atoms [3,6] in cl 48 in P28 using law 1 (...P30)
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
54. frontier1([tree(A,B)|C],D) :- frontier(A,E), frontier(B,F), append(E,G,D),
    frontierlist(C,H), flatten(H,I), append(F,I,G).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).

P31: by folding cl 54 in P30 using cl 8 (...P32)
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
56. frontier1([tree(A,B)|C],D) :- frontier(A,E), frontier(B,F),
    frontierlist(C,G), flatten([F|G],H), append(E,H,D).
```

```
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).

P33: by folding cl 56 in P32 using cl 8 (...P34)
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
58. frontier1([tree(A,B)|C],D) :- frontier(B,E), frontierlist(C,F),
    flatten([G,E|F],D), frontier(A,G).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).

P35: by folding cl 58 in P34 using cl 6 (...P36)
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
60. frontier1([tree(A,B)|C],D) :- frontier(A,E), frontierlist([B|C],[F|G]),
    flatten([E,F|G],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).

P37: by folding cl 60 in P36 using cl 6
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
61. frontier1([tree(A,B)|C],D) :- frontierlist([A,B|C],[E,F|G]),
    flatten([E,F|G],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).

P38: by replacing atoms [1,2] in cl 61 in P37 using law 5
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
62. frontier1([tree(A,B)|C],D) :- frontierlist([A,B|C],E), flatten(E,D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).

P39: by folding cl 62 in P38 using cl 4
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).

P40: by unfolding cl 9 in P39 wrt atom 1 using P1
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
64. eqtreelist(A,B) :- frontierlist(A,C), flatten(C,D), frontier1(B,D).
```

```
P41: by unfolding cl 64 in P40 wrt atom 1 using P1
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
65. eqtreelist([],A) :- flatten([],B), frontier1(A,B).
66. eqtreelist([A|B],C) :- frontier(A,D), frontierlist(B,E), flatten([D|E],F),
    frontier1(C,F).

P42: by unfolding cl 65 in P41 wrt atom 1 using P1
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
67. eqtreelist([],A) :- frontier1(A,[]).
66. eqtreelist([A|B],C) :- frontier(A,D), frontierlist(B,E), flatten([D|E],F),
    frontier1(C,F).

P43: by unfolding cl 66 in P42 wrt atom 3 using P1
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
67. eqtreelist([],A) :- frontier1(A,[]).
68. eqtreelist([A|B],C) :- frontier(A,D), frontierlist(B,E), flatten(E,F),
    append(D,F,G), frontier1(C,G).

P44: by unfolding cl 68 in P43 wrt atom 1 using P1
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
67. eqtreelist([],A) :- frontier1(A,[]).
69. eqtreelist([tree(A,B)|C],D) :- frontier(A,E), frontier(B,F), append(E,F,G),
    frontierlist(C,H), flatten(H,I), append(G,I,J), frontier1(D,J).
70. eqtreelist([tip(A)|B],C) :- frontierlist(B,D), flatten(D,E),
    append([A],E,F), frontier1(C,F).

P45: by unfolding cl 70 in P44 wrt atom 3 using P1
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
```

```
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
67. eqtreelist([],A) :- frontier1(A,[]).
69. eqtreelist([tree(A,B)|C],D) :- frontier(A,E), frontier(B,F), append(E,F,G),
    frontierlist(C,H), flatten(H,I), append(G,I,J), frontier1(D,J).
71. eqtreelist([tip(A)|B],C) :- frontierlist(B,D), flatten(D,E),
    append([],E,F), frontier1(C,[A|F]).

P46: by unfolding cl 71 in P45 wrt atom 3 using P1
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
67. eqtreelist([],A) :- frontier1(A,[]).
69. eqtreelist([tree(A,B)|C],D) :- frontier(A,E), frontier(B,F), append(E,F,G),
    frontierlist(C,H), flatten(H,I), append(G,I,J), frontier1(D,J).
72. eqtreelist([tip(A)|B],C) :- frontierlist(B,D), flatten(D,E),
    frontier1(C,[A|E]).

P47: by folding cl 72 in P46 using cl 4
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
67. eqtreelist([],A) :- frontier1(A,[]).
69. eqtreelist([tree(A,B)|C],D) :- frontier(A,E), frontier(B,F), append(E,F,G),
    frontierlist(C,H), flatten(H,I), append(G,I,J), frontier1(D,J).
74. eqtreelist([tip(A)|B],C) :- frontier1(B,D), frontier1(C,[A|D]).

P48: by replacing atoms [3,6] in cl 69 in P47 using law 1 (...P49)
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
67. eqtreelist([],A) :- frontier1(A,[]).
76. eqtreelist([tree(A,B)|C],D) :- frontier(A,E), frontier(B,F), append(E,G,H),
    frontierlist(C,I), flatten(I,J), append(F,J,G), frontier1(D,H).
74. eqtreelist([tip(A)|B],C) :- frontier1(B,D), frontier1(C,[A|D]).

P50: by folding cl 76 in P49 using cl 8 (...P51)
43. eqtree(A,B) :- eqtreelist([A],[B]).
```

```
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
67. eqtreelist([],A) :- frontier1(A,[]).
78. eqtreelist([tree(A,B)|C],D) :- frontier(A,E), frontier(B,F),
    frontierlist(C,G), flatten([F|G],H), append(E,H,I), frontier1(D,I).
74. eqtreelist([tip(A)|B],C) :- frontier1(B,D), frontier1(C,[A|D]).

P52: by folding cl 78 in P51 using cl 8 (...P53)
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
67. eqtreelist([],A) :- frontier1(A,[]).
80. eqtreelist([tree(A,B)|C],D) :- frontier(B,E), frontierlist(C,F),
    flatten([G,E|F],H), frontier1(D,H), frontier(A,G).
74. eqtreelist([tip(A)|B],C) :- frontier1(B,D), frontier1(C,[A|D]).

P54: by folding cl 80 in P53 using cl 6 (...P55)
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
67. eqtreelist([],A) :- frontier1(A,[]).
82. eqtreelist([tree(A,B)|C],D) :- frontier(A,E), frontierlist([B|C],[F|G]),
    flatten([E,F|G],H), frontier1(D,H).
74. eqtreelist([tip(A)|B],C) :- frontier1(B,D), frontier1(C,[A|D]).

P56: by folding cl 82 in P55 using cl 6
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
67. eqtreelist([],A) :- frontier1(A,[]).
83. eqtreelist([tree(A,B)|C],D) :- frontierlist([A,B|C],[E,F|G]),
    flatten([E,F|G],H), frontier1(D,H).
74. eqtreelist([tip(A)|B],C) :- frontier1(B,D), frontier1(C,[A|D]).

P57: by replacing atoms [1,2] in cl 83 in P56 using law 5
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
```

```
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
67. eqtreelist([],A) :- frontier1(A,[]).
84. eqtreelist([tree(A,B)|C],D) :- frontierlist([A,B|C],E), flatten(E,F),
    frontier1(D,F).
74. eqtreelist([tip(A)|B],C) :- frontier1(B,D), frontier1(C,[A|D]).

P58: by folding cl 84 in P57 using cl 4
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
67. eqtreelist([],A) :- frontier1(A,[]).
85. eqtreelist([tree(A,B)|C],D) :- frontier1([A,B|C],E), frontier1(D,E).
74. eqtreelist([tip(A)|B],C) :- frontier1(B,D), frontier1(C,[A|D]).

P59: by folding cl 85 in P58 using cl 9
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
67. eqtreelist([],A) :- frontier1(A,[]).
86. eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
74. eqtreelist([tip(A)|B],C) :- frontier1(B,D), frontier1(C,[A|D]).

P60: by adding def 87 to P59
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
67. eqtreelist([],A) :- frontier1(A,[]).
86. eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
74. eqtreelist([tip(A)|B],C) :- frontier1(B,D), frontier1(C,[A|D]).
87. eqtreelist1(A,B,C) :- frontier1(B,D), frontier1(C,[A|D]).

P61: by folding cl 74 in P60 using cl 87
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
67. eqtreelist([],A) :- frontier1(A,[]).
```

```
86. eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88. eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
87. eqtreelist1(A,B,C) :- frontier1(B,D), frontier1(C,[A|D]).

P62: by unfolding cl 67 in P61 wrt atom 1 using P1
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
89. eqtreelist([],A) :- frontierlist(A,B), flatten(B,[]).
86. eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88. eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
87. eqtreelist1(A,B,C) :- frontier1(B,D), frontier1(C,[A|D]).

P63: by unfolding cl 89 in P62 wrt atom 1 using P1
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
90. eqtreelist([],[]) :- flatten([],[]).
91. eqtreelist([],[A|B]) :- frontier(A,C), frontierlist(B,D), flatten([C|D],[]).
86. eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88. eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
87. eqtreelist1(A,B,C) :- frontier1(B,D), frontier1(C,[A|D]).

P64: by unfolding cl 90 in P63 wrt atom 1 using P1
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92. eqtreelist([],[]).
91. eqtreelist([],[A|B]) :- frontier(A,C), frontierlist(B,D), flatten([C|D],[]).
86. eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88. eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
87. eqtreelist1(A,B,C) :- frontier1(B,D), frontier1(C,[A|D]).

P65: by unfolding cl 91 in P64 wrt atom 3 using P1
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
```

```
92. eqtreelist([],[]).
93. eqtreelist([],[A|B]) :- frontier(A,C), frontierlist(B,D), flatten(D,E),
    append(C,E,[]).
86. eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88. eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
87. eqtreelist1(A,B,C) :- frontier1(B,D), frontier1(C,[A|D]).

P66: by unfolding cl 93 in P65 wrt atom 1 using P1 (...P68)
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92. eqtreelist([],[]).
94. eqtreelist([],[tree(A,B)|C]) :- frontier(A,D), frontier(B,E),
    append(D,E,F), frontierlist(C,G), flatten(G,H), append(F,H,[]).
95. eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
    append([A],D,[]).
86. eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88. eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
87. eqtreelist1(A,B,C) :- frontier1(B,D), frontier1(C,[A|D]).

P67: by replacing atoms [3,6] in cl 94 in P66 using law 1 (...P68)
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92. eqtreelist([],[]).
97. eqtreelist([],[tree(A,B)|C]) :- frontier(A,D), frontier(B,E),
    append(D,F,[]), frontierlist(C,G), flatten(G,H), append(E,H,F).
95. eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
    append([A],D,[]).
86. eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88. eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
87. eqtreelist1(A,B,C) :- frontier1(B,D), frontier1(C,[A|D]).

P69: by folding cl 97 in P68 using cl 8 (...P70)
43. eqtree(A,B) :- eqtreelist([A],[B]).
46. frontier1([],[]).
63. frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52. frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92. eqtreelist([],[]).
```

```
99. eqtreelist([],[tree(A,B)|C]) :- frontier(A,D), frontier(B,E),
    frontierlist(C,F), flatten([E|F],G), append(D,G,[]).
95. eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
    append([A],D,[]).
86. eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88. eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
87. eqtreelist1(A,B,C) :- frontier1(B,D), frontier1(C,[A|D]).

P71: by folding cl 99 in P70 using cl 8 (...P72)
43.  eqtree(A,B) :- eqtreelist([A],[B]).
46.  frontier1([],[]).
63.  frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52.  frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92.  eqtreelist([],[]).
101. eqtreelist([],[tree(A,B)|C]) :- frontier(B,D), frontierlist(C,E),
     flatten([F,D|E],[]), frontier(A,F).
95.  eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
     append([A],D,[]).
86.  eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88.  eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
87.  eqtreelist1(A,B,C) :- frontier1(B,D), frontier1(C,[A|D]).

P73: by folding cl 101 in P72 using cl 6 (...P74)
43.  eqtree(A,B) :- eqtreelist([A],[B]).
46.  frontier1([],[]).
63.  frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52.  frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92.  eqtreelist([],[]).
103. eqtreelist([],[tree(A,B)|C]) :- frontier(A,D),
     frontierlist([B|C],[E|F]), flatten([D,E|F],[]).
95.  eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
     append([A],D,[]).
86.  eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88.  eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
87.  eqtreelist1(A,B,C) :- frontier1(B,D), frontier1(C,[A|D]).

P75: by folding cl 103 in P74 using cl 6
43.  eqtree(A,B) :- eqtreelist([A],[B]).
46.  frontier1([],[]).
63.  frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52.  frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
```

```
92.  eqtreelist([],[]).
104. eqtreelist([],[tree(A,B)|C]) :- frontierlist([A,B|C],[D,E|F]),
     flatten([D,E|F],[]).
95.  eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
     append([A],D,[]).
86.  eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88.  eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
87.  eqtreelist1(A,B,C) :- frontier1(B,D), frontier1(C,[A|D]).

P76: by replacing atoms [1,2] in cl 104 in P75 using law 5
43.  eqtree(A,B) :- eqtreelist([A],[B]).
46.  frontier1([],[]).
63.  frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52.  frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92.  eqtreelist([],[]).
105. eqtreelist([],[tree(A,B)|C]) :- frontierlist([A,B|C],D), flatten(D,[]).
95.  eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
     append([A],D,[]).
86.  eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88.  eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
87.  eqtreelist1(A,B,C) :- frontier1(B,D), frontier1(C,[A|D]).

P77: by folding cl 105 in P76 using cl 4
43.  eqtree(A,B) :- eqtreelist([A],[B]).
46.  frontier1([],[]).
63.  frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52.  frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92.  eqtreelist([],[]).
106. eqtreelist([],[tree(A,B)|C]) :- frontier1([A,B|C],[]).
95.  eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
     append([A],D,[]).
86.  eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88.  eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
87.  eqtreelist1(A,B,C) :- frontier1(B,D), frontier1(C,[A|D]).

P78: by unfolding cl 87 in P77 wrt atom 2 using P1
43.  eqtree(A,B) :- eqtreelist([A],[B]).
46.  frontier1([],[]).
63.  frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52.  frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92.  eqtreelist([],[]).
```

```
106. eqtreelist([],[tree(A,B)|C]) :- frontier1([A,B|C],[]).
95.  eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
     append([A],D,[]).
86.  eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88.  eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
107. eqtreelist1(A,B,C) :- frontier1(B,D), frontierlist(C,E),
     flatten(E,[A|D]).

P79: by unfolding cl 107 in P78 wrt atom 2 using P1
43.  eqtree(A,B) :- eqtreelist([A],[B]).
46.  frontier1([],[]).
63.  frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52.  frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92.  eqtreelist([],[]).
106. eqtreelist([],[tree(A,B)|C]) :- frontier1([A,B|C],[]).
95.  eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
     append([A],D,[]).
86.  eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88.  eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
108. eqtreelist1(A,B,[]) :- frontier1(B,C), flatten([],[A|C]).
109. eqtreelist1(A,B,[C|D]) :- frontier1(B,E), frontier(C,F),
     frontierlist(D,G), flatten([F|G],[A|E]).

P80: by unfolding cl 109 in P79 wrt atom 4 using P1
43.  eqtree(A,B) :- eqtreelist([A],[B]).
46.  frontier1([],[]).
63.  frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52.  frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92.  eqtreelist([],[]).
106. eqtreelist([],[tree(A,B)|C]) :- frontier1([A,B|C],[]).
95.  eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
     append([A],D,[]).
86.  eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88.  eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
108. eqtreelist1(A,B,[]) :- frontier1(B,C), flatten([],[A|C]).
110. eqtreelist1(A,B,[C|D]) :- frontier1(B,E), frontier(C,F),
     frontierlist(D,G), flatten(G,H), append(F,H,[A|E]).

P81: by unfolding cl 110 in P80 wrt atom 2 using P1
43.  eqtree(A,B) :- eqtreelist([A],[B]).
46.  frontier1([],[]).
```

```
63.  frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52.  frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92.  eqtreelist([],[]).
106. eqtreelist([],[tree(A,B)|C]) :- frontier1([A,B|C],[]).
95.  eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
     append([A],D,[]).
86.  eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88.  eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
108. eqtreelist1(A,B,[]) :- frontier1(B,C), flatten([],[A|C]).
111. eqtreelist1(A,B,[tree(C,D)|E]) :- frontier1(B,F), frontier(C,G),
     frontier(D,H), append(G,H,I), frontierlist(E,J), flatten(J,K),
     append(I,K,[A|F]).
112. eqtreelist1(A,B,[tip(C)|D]) :- frontier1(B,E), frontierlist(D,F),
     flatten(F,G), append([C],G,[A|E]).

P82: by unfolding cl 112 in P81 wrt atom 4 using P1
43.  eqtree(A,B) :- eqtreelist([A],[B]).
46.  frontier1([],[]).
63.  frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52.  frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92.  eqtreelist([],[]).
106. eqtreelist([],[tree(A,B)|C]) :- frontier1([A,B|C],[]).
95.  eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
     append([A],D,[]).
86.  eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88.  eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
108. eqtreelist1(A,B,[]) :- frontier1(B,C), flatten([],[A|C]).
111. eqtreelist1(A,B,[tree(C,D)|E]) :- frontier1(B,F), frontier(C,G),
     frontier(D,H), append(G,H,I), frontierlist(E,J), flatten(J,K),
     append(I,K,[A|F]).
113. eqtreelist1(A,B,[tip(A)|C]) :- frontier1(B,D), frontierlist(C,E),
     flatten(E,F), append([],F,D).

P83: by unfolding cl 113 in P82 wrt atom 4 using P1
43.  eqtree(A,B) :- eqtreelist([A],[B]).
46.  frontier1([],[]).
63.  frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52.  frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92.  eqtreelist([],[]).
106. eqtreelist([],[tree(A,B)|C]) :- frontier1([A,B|C],[]).
95.  eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
```

```
     append([A],D,[]).
86.  eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88.  eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
108. eqtreelist1(A,B,[]) :- frontier1(B,C), flatten([],[A|C]).
111. eqtreelist1(A,B,[tree(C,D)|E]) :- frontier1(B,F), frontier(C,G),
     frontier(D,H), append(G,H,I), frontierlist(E,J), flatten(J,K),
     append(I,K,[A|F]).
114. eqtreelist1(A,B,[tip(A)|C]) :- frontier1(B,D), frontierlist(C,E),
     flatten(E,D).

P84: by folding cl 114 in P83 using cl 4
43.  eqtree(A,B) :- eqtreelist([A],[B]).
46.  frontier1([],[]).
63.  frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52.  frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92.  eqtreelist([],[]).
106. eqtreelist([],[tree(A,B)|C]) :- frontier1([A,B|C],[]).
95.  eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
     append([A],D,[]).
86.  eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88.  eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
108. eqtreelist1(A,B,[]) :- frontier1(B,C), flatten([],[A|C]).
111. eqtreelist1(A,B,[tree(C,D)|E]) :- frontier1(B,F), frontier(C,G),
     frontier(D,H), append(G,H,I), frontierlist(E,J), flatten(J,K),
     append(I,K,[A|F]).
115. eqtreelist1(A,B,[tip(A)|C]) :- frontier1(B,D), frontier1(C,D).

P85: by replacing atoms [4,7] in cl 111 in P84 using law 1 (...P86)
43.  eqtree(A,B) :- eqtreelist([A],[B]).
46.  frontier1([],[]).
63.  frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52.  frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92.  eqtreelist([],[]).
106. eqtreelist([],[tree(A,B)|C]) :- frontier1([A,B|C],[]).
95.  eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
     append([A],D,[]).
86.  eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88.  eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
108. eqtreelist1(A,B,[]) :- frontier1(B,C), flatten([],[A|C]).
117. eqtreelist1(A,B,[tree(C,D)|E]) :- frontier1(B,F), frontier(C,G),
     frontier(D,H), append(G,I,[A|F]), frontierlist(E,J), flatten(J,K),
```

```
     append(H,K,I).
115. eqtreelist1(A,B,[tip(A)|C]) :- frontier1(B,D), frontier1(C,D).

P87: by folding cl 117 in P86 using cl 8 (...P88)
43.  eqtree(A,B) :- eqtreelist([A],[B]).
46.  frontier1([],[]).
63.  frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52.  frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92.  eqtreelist([],[]).
106. eqtreelist([],[tree(A,B)|C]) :- frontier1([A,B|C],[]).
95.  eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
     append([A],D,[]).
86.  eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88.  eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
108. eqtreelist1(A,B,[]) :- frontier1(B,C), flatten([],[A|C]).
119. eqtreelist1(A,B,[tree(C,D)|E]) :- frontier1(B,F), frontier(C,G),
     frontier(D,H), frontierlist(E,I), flatten([H|I],J), append(G,J,[A|F]).
115. eqtreelist1(A,B,[tip(A)|C]) :- frontier1(B,D), frontier1(C,D).

P89: by folding cl 119 in P88 using cl 6
43.  eqtree(A,B) :- eqtreelist([A],[B]).
46.  frontier1([],[]).
63.  frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52.  frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92.  eqtreelist([],[]).
106. eqtreelist([],[tree(A,B)|C]) :- frontier1([A,B|C],[]).
95.  eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
     append([A],D,[]).
86.  eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88.  eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
108. eqtreelist1(A,B,[]) :- frontier1(B,C), flatten([],[A|C]).
120. eqtreelist1(A,B,[tree(C,D)|E]) :- frontier1(B,F), frontier(C,G),
     frontierlist([D|E],[H|I]), flatten([H|I],J), append(G,J,[A|F]).
115. eqtreelist1(A,B,[tip(A)|C]) :- frontier1(B,D), frontier1(C,D).

P90: by folding cl 120 in P89 using cl 8
43.  eqtree(A,B) :- eqtreelist([A],[B]).
46.  frontier1([],[]).
63.  frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52.  frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92.  eqtreelist([],[]).
```

```
106. eqtreelist([],[tree(A,B)|C]) :- frontier1([A,B|C],[]).
95.  eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
     append([A],D,[]).
86.  eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88.  eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
108. eqtreelist1(A,B,[]) :- frontier1(B,C), flatten([],[A|C]).
121. eqtreelist1(A,B,[tree(C,D)|E]) :- frontier1(B,F), frontier(C,G),
     frontierlist([D|E],[H|I]), flatten([G,H|I],[A|F]).
115. eqtreelist1(A,B,[tip(A)|C]) :- frontier1(B,D), frontier1(C,D).

P91: by folding cl 121 in P90 using cl 6
43.  eqtree(A,B) :- eqtreelist([A],[B]).
46.  frontier1([],[]).
63.  frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52.  frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92.  eqtreelist([],[]).
106. eqtreelist([],[tree(A,B)|C]) :- frontier1([A,B|C],[]).
95.  eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
     append([A],D,[]).
86.  eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88.  eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
108. eqtreelist1(A,B,[]) :- frontier1(B,C), flatten([],[A|C]).
122. eqtreelist1(A,B,[tree(C,D)|E]) :- frontier1(B,F),
     frontierlist([C,D|E],[G,H|I]), flatten([G,H|I],[A|F]).
115. eqtreelist1(A,B,[tip(A)|C]) :- frontier1(B,D), frontier1(C,D).

P92: by replacing atoms [2,3] in cl 122 in P91 using law 5
43.  eqtree(A,B) :- eqtreelist([A],[B]).
46.  frontier1([],[]).
63.  frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52.  frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92.  eqtreelist([],[]).
106. eqtreelist([],[tree(A,B)|C]) :- frontier1([A,B|C],[]).
95.  eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
     append([A],D,[]).
86.  eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88.  eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
108. eqtreelist1(A,B,[]) :- frontier1(B,C), flatten([],[A|C]).
123. eqtreelist1(A,B,[tree(C,D)|E]) :- frontier1(B,F),
     frontierlist([C,D|E],G), flatten(G,[A|F]).
115. eqtreelist1(A,B,[tip(A)|C]) :- frontier1(B,D), frontier1(C,D).
```

```
P93: by folding cl 123 in P92 using cl 4
43.  eqtree(A,B) :- eqtreelist([A],[B]).
46.  frontier1([],[]).
63.  frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52.  frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92.  eqtreelist([],[]).
106. eqtreelist([],[tree(A,B)|C]) :- frontier1([A,B|C],[]).
95.  eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
     append([A],D,[]).
86.  eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88.  eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
108. eqtreelist1(A,B,[]) :- frontier1(B,C), flatten([],[A|C]).
124. eqtreelist1(A,B,[tree(C,D)|E]) :- frontier1(B,F),
     frontier1([C,D|E],[A|F]).
115. eqtreelist1(A,B,[tip(A)|C]) :- frontier1(B,D), frontier1(C,D).

P94: by folding cl 115 in P93 using cl 9
43.  eqtree(A,B) :- eqtreelist([A],[B]).
46.  frontier1([],[]).
63.  frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52.  frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92.  eqtreelist([],[]).
106. eqtreelist([],[tree(A,B)|C]) :- frontier1([A,B|C],[]).
95.  eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
     append([A],D,[]).
86.  eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88.  eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
108. eqtreelist1(A,B,[]) :- frontier1(B,C), flatten([],[A|C]).
124. eqtreelist1(A,B,[tree(C,D)|E]) :- frontier1(B,F),
     frontier1([C,D|E],[A|F]).
125. eqtreelist1(A,B,[tip(A)|C]) :- eqtreelist(B,C).

P95: by folding cl 124 in P94 using cl 87
43.  eqtree(A,B) :- eqtreelist([A],[B]).
46.  frontier1([],[]).
63.  frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52.  frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92.  eqtreelist([],[]).
106. eqtreelist([],[tree(A,B)|C]) :- frontier1([A,B|C],[]).
95.  eqtreelist([],[tip(A)|B]) :- frontierlist(B,C), flatten(C,D),
     append([A],D,[]).
```

```
86.  eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88.  eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
108. eqtreelist1(A,B,[]) :- frontier1(B,C), flatten([],[A|C]).
126. eqtreelist1(A,B,[tree(C,D)|E]) :- eqtreelist1(A,B,[C,D|E]).
125. eqtreelist1(A,B,[tip(A)|C]) :- eqtreelist(B,C).

P96: by deleting cl 95 in P95
43.  eqtree(A,B) :- eqtreelist([A],[B]).
46.  frontier1([],[]).
63.  frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52.  frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92.  eqtreelist([],[]).
106. eqtreelist([],[tree(A,B)|C]) :- frontier1([A,B|C],[]).
86.  eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88.  eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
108. eqtreelist1(A,B,[]) :- frontier1(B,C), flatten([],[A|C]).
126. eqtreelist1(A,B,[tree(C,D)|E]) :- eqtreelist1(A,B,[C,D|E]).
125. eqtreelist1(A,B,[tip(A)|C]) :- eqtreelist(B,C).

P97: by deleting cl 108 in P96
43.  eqtree(A,B) :- eqtreelist([A],[B]).
46.  frontier1([],[]).
63.  frontier1([tree(A,B)|C],D) :- frontier1([A,B|C],D).
52.  frontier1([tip(A)|B],[A|C]) :- frontier1(B,C).
92.  eqtreelist([],[]).
106. eqtreelist([],[tree(A,B)|C]) :- frontier1([A,B|C],[]).
86.  eqtreelist([tree(A,B)|C],D) :- eqtreelist([A,B|C],D).
88.  eqtreelist([tip(A)|B],C) :- eqtreelist1(A,B,C).
126. eqtreelist1(A,B,[tree(C,D)|E]) :- eqtreelist1(A,B,[C,D|E]).
125. eqtreelist1(A,B,[tip(A)|C]) :- eqtreelist(B,C).
```

The result of this transformation sequence is Program P97. We have obtained the following correctness proof:

```
| ?- current_constraint_system.

Clause weights:

[(1,w(1)),(2,w(2)),(43,w(40)),(46,w(43)),(63,w(56)),(52,w(49)),(5,w(5)),
 (6,w(6)),(7,w(7)),(8,w(8)),(92,w(80)),(106,w(90)),(86,w(74)),(88,w(76)),
 (10,w(10)),(11,w(11)),(12,w(12)),(13,w(13)),(14,w(14)),(15,w(15)),
```

```
 (16,w(16)),(17,w(17)),(18,w(18)),(19,w(19)),(20,w(20)),(21,w(21)),
 (22,w(22)),(23,w(23)),(126,w(108)),(125,w(107))]

The Correctness Constraint System for the Current Program is:

[w(24)=:=w(3)+(w(17)-w(16)),w(10)+w(11)-w(16)+w(17)=<0,w(14)=w(15),
 w(25)=:=w(24)+(w(17)-w(16)),w(10)+w(11)-w(16)+w(17)=<0,w(14)=w(15),
 w(26)=<w(25)-w(7),w(27)=<w(26)-w(7),w(28)=:=w(27)+(w(21)-w(20)),w(14)=w(15),
 w(20)-w(21)>=0,w(29)=:=w(28)+(w(21)-w(20)),w(14)=w(15),w(20)-w(21)>=0,
 w(30)=<w(29)-w(8),w(31)=<w(30)-w(8),w(32)=<w(31)-w(5),w(33)=<w(32)-w(5),
 w(34)=<w(33)-w(6),w(35)=<w(34)-w(6),w(36)=:=w(35)+(w(19)-w(18)),w(14)=w(15),
 w(18)-w(19)>=0,w(37)=:=w(36)+(w(19)-w(18)),w(14)=w(15),w(18)-w(19)>=0,
 w(38)=<w(37)-w(4),w(39)=<w(38)-w(4),w(40)=<w(39)-w(9),w(41)=:=w(4)+w(5),
 w(42)=:=w(4)+w(6),w(43)=:=w(41)+w(7),w(44)=:=w(42)+w(8),w(45)=:=w(44)+w(1),
 w(46)=:=w(44)+w(2),w(47)=:=w(46)+w(11),w(48)=:=w(47)+w(10),w(49)=<w(48)-w(4),
 w(50)=:=w(45)+(w(15)-w(14)),w(14)=w(15),w(51)=<w(50)-w(8),w(52)=<w(51)-w(8),
 w(53)=<w(52)-w(6),w(54)=<w(53)-w(6),w(55)=:=w(54)+(w(23)-w(22)),w(22)-w(23)>=0,
 w(56)=<w(55)-w(4),w(57)=:=w(9)+w(4),w(58)=:=w(57)+w(5),w(59)=:=w(57)+w(6),
 w(60)=:=w(58)+w(7),w(61)=:=w(59)+w(8),w(62)=:=w(61)+w(1),w(63)=:=w(61)+w(2),
 w(64)=:=w(63)+w(11),w(65)=:=w(64)+w(10),w(66)=<w(65)-w(4),w(68)=<w(67)-w(8),
 w(67)=:=w(62)+(w(15)-w(14)),w(69)=<w(68)-w(8),w(14)=w(15), w(70)=<w(69)-w(6),
 w(71)=<w(70)-w(6),w(72)=:=w(71)+(w(23)-w(22)),w(22)-w(23)>=0,w(73)=<w(72)-w(4),
 w(74)=<w(73)-w(9),w(76)=<w(66)-w(75),w(77)=:=w(60)+w(4),w(78)=:=w(77)+w(5),
 w(79)=:=w(77)+w(6),w(80)=:=w(78)+w(7),w(81)=:=w(79)+w(8),w(82)=:=w(81)+w(1),
 w(83)=:=w(81)+w(2),w(84)=:=w(82)+(w(15)-w(14)),w(14)=w(15),w(85)=<w(84)-w(8),
 w(86)=<w(85)-w(8),w(87)=<w(86)-w(6),w(88)=<w(87)-w(6),w(22)-w(23)>=0,
 w(89)=:=w(88)+(w(23)-w(22)),w(90)=<w(89)-w(4),w(91)=:=w(75)+w(4),w(14)=w(15),
 w(92)=:=w(91)+w(5),w(94)=:=w(93)+w(8),w(95)=:=w(94)+w(1),w(96)=:=w(94)+w(2),
 w(97)=:=w(96)+w(11),w(98)=:=w(97)+w(10),w(99)=<w(98)-w(4),w(93)=:=w(91)+w(6),
 w(100)=:=w(95)+(w(15)-w(14)),w(101)=<w(100)-w(8),w(102)=<w(101)-w(6),
 w(103)=<w(102)-w(8),w(104)=<w(103)-w(6),w(22)-w(23)>=0,w(106)=<w(105)-w(4),
 w(105)=:=w(104)+(w(23)-w(22)),w(107)=<w(99)-w(9),w(108)=<w(106)-w(75)]

| ?- solve_constraint_system.
The Correctness Constraint System for the Current Program is SATISFIABLE.

A solution is:
Clause number 1 weight 3
Clause number 2 weight 1
Clause number 43 weight 1
Clause number 46 weight 2
```

```
Clause number 63 weight 1
Clause number 52 weight 1
Clause number 5 weight 1
Clause number 6 weight 1
Clause number 7 weight 1
Clause number 8 weight 1
Clause number 92 weight 4
Clause number 106 weight 1
Clause number 86 weight 1
Clause number 88 weight 1
Clause number 10 weight 1
Clause number 11 weight 1
Clause number 12 weight 1
Clause number 13 weight 1
Clause number 14 weight 1
Clause number 15 weight 1
Clause number 16 weight 3
Clause number 17 weight 1
Clause number 18 weight 1
Clause number 19 weight 1
Clause number 20 weight 1
Clause number 21 weight 1
Clause number 22 weight 1
Clause number 23 weight 1
Clause number 126 weight 1
Clause number 125 weight 1

Elapsed time 370ms
```

Chapter 7

Appendix B

Below we give the definitions of the array formulas $genc_2$ through $genc_8$ occurring in the program R: Chapter 5, Section 5.3.

$$
\begin{array}{l}
genc_2(s(P,J,Q,S)) \equiv_{def} \\
\quad \exists i,l(rd(P,i,\lambda) \ \wedge \ l{>}1 \ \wedge \ rd(J,i,1) \ \wedge \ rd(Q,i,1) \ \wedge \ rd(S,1,i) \wedge \\
\qquad \forall k(1{\leq}k{\leq}l \ \rightarrow \ ((rd(P,k,ncs) \ \wedge \ rd(Q,k,0)) \vee \\
\qquad\qquad\qquad\qquad (rd(P,k,w) \ \wedge \ rd(J,k,1) \ \wedge \ rd(Q,k,0)) \vee \\
\qquad\qquad\qquad\qquad (rd(P,k,\lambda) \ \wedge \ rd(J,k,1) \ \wedge \ rd(Q,k,1)))) \wedge \\
\qquad ln(P,l) \ \wedge \ ln(J,l) \ \wedge \ ln(Q,l) \ \wedge \ ln(S,l))
\end{array}
$$

$$
\begin{array}{l}
genc_3(s(P,J,Q,S)) \equiv_{def} \\
\quad \exists i,k,l(2{\leq}k{<}l \wedge ((rd(P,i,w) \wedge rd(J,i,k{+}1) \wedge rd(Q,i,k) \wedge rd(S,k,i)) \vee \\
\qquad\qquad\qquad (rd(P,i,\lambda) \wedge rd(J,i,k) \wedge rd(Q,i,k) \wedge rd(S,k,i))) \wedge \\
\qquad\quad \forall j((1{\leq}j{\leq}l \ \wedge \neg(j{=}i)) \ \rightarrow \ ((rd(P,j,ncs) \wedge rd(Q,j,0)) \vee \\
\qquad\qquad\qquad\qquad\qquad\qquad (rd(P,j,w) \wedge rd(J,j,1) \wedge rd(Q,j,0)))) \wedge \\
\qquad\quad ln(P,l) \ \wedge \ ln(J,l) \ \wedge \ ln(Q,l) \ \wedge \ ln(S,l))
\end{array}
$$

$$
\begin{aligned}
&genc_4(s(P,J,Q,S)) \equiv_{def} \\
&\quad \exists m,l(1 \leq m \leq l \ \wedge \ \forall k(1 \leq k < m \ \rightarrow \\
&\qquad\qquad \exists i(rd(P,i,\lambda) \wedge rd(J,i,k) \wedge rd(Q,i,k) \wedge rd(S,k,i))) \wedge \\
&\qquad \forall j(1 \leq j \leq l \rightarrow \ ((rd(P,j,ncs) \wedge rd(Q,j,0)) \vee \\
&\qquad\qquad \exists k(1 \leq k < m \wedge ((rd(P,j,w) \wedge rd(J,j,k+1) \wedge rd(Q,j,k)) \vee \\
&\qquad\qquad\qquad (rd(P,j,\lambda) \wedge rd(J,j,k) \wedge rd(Q,j,k)))))) \wedge \\
&\qquad ln(P,l) \ \wedge \ ln(J,l) \ \wedge \ ln(Q,l) \ \wedge \ ln(S,l))
\end{aligned}
$$

$$
\begin{aligned}
&genc_5(s(P,J,Q,S)) \equiv_{def} \\
&\quad \exists i,k,l(2 \leq k < l \wedge ((rd(P,i,w) \wedge rd(J,i,k+1) \wedge rd(Q,i,k) \wedge rd(S,k,i)) \vee \\
&\qquad\qquad (rd(P,i,\lambda) \wedge rd(J,i,k) \wedge rd(Q,i,k) \wedge rd(S,k,i))) \wedge \\
&\qquad \exists m(1 \leq m \leq k \wedge \\
&\qquad\quad \forall u(1 \leq u \leq m \ \rightarrow \\
&\qquad\qquad \exists j(rd(P,j,\lambda) \wedge rd(J,j,u) \wedge rd(Q,j,u) \wedge rd(S,u,j))) \wedge \\
&\qquad\quad \forall n((1 \leq n \leq l \wedge \neg(n = i)) \rightarrow \\
&\qquad\qquad ((rd(P,n,ncs) \wedge rd(Q,n,0)) \vee \\
&\qquad\qquad\ \exists r(1 \leq r \leq m \wedge \\
&\qquad\qquad\qquad ((rd(P,n,w) \wedge rd(J,n,r+1) \wedge rd(Q,n,r)) \vee \\
&\qquad\qquad\qquad\ (rd(P,n,\lambda) \wedge rd(J,n,r) \wedge rd(Q,n,r))))))) \wedge \\
&\qquad ln(P,l) \ \wedge \ ln(J,l) \ \wedge \ ln(Q,l) \ \wedge \ ln(S,l))
\end{aligned}
$$

$$
\begin{aligned}
&genc_6(s(P,J,Q,S)) \equiv_{def} \\
&\quad \exists i,l,u(rd(P,i,cs) \wedge rd(J,i,u+1) \wedge rd(Q,i,u) \wedge rd(S,u,i) \ \wedge \ u+1 = l \wedge \\
&\qquad \forall j((1 \leq j \leq l \wedge \neg(j = i)) \rightarrow ((rd(P,k,ncs) \wedge rd(Q,k,0)) \vee \\
&\qquad\qquad\qquad (rd(P,k,w) \wedge rd(J,k,1) \wedge rd(Q,k,0)))) \wedge \\
&\qquad ln(P,l) \ \wedge \ ln(J,l) \ \wedge \ ln(Q,l) \ \wedge \ ln(S,l))
\end{aligned}
$$

$$
\begin{aligned}
&genc_7(s(P,J,Q,S)) \equiv_{def} \\
&\quad \exists i,l,u(rd(P,i,cs) \wedge rd(J,i,u+1) \wedge rd(Q,i,u) \ \wedge \ u+1 = l \wedge \\
&\qquad \forall k(1 \leq k < l \ \rightarrow \ \exists i(rd(P,i,\lambda) \wedge rd(J,i,k) \wedge rd(Q,i,k) \wedge rd(S,k,i))) \wedge \\
&\qquad ln(P,l) \ \wedge \ ln(J,l) \ \wedge \ ln(Q,l) \ \wedge \ ln(S,l))
\end{aligned}
$$

$$
\begin{array}{l}
genc_8(s(P,J,Q,S)) \equiv_{def} \\
\quad \exists i,l,u(rd(P,i,cs) \wedge rd(J,i,u{+}1) \wedge rd(Q,i,u) \ \wedge \ u{+}1{=}l \ \wedge \\
\qquad \exists m(1{\leq}m{\leq}l \wedge \forall n(1{\leq}n{<}m \ \rightarrow \\
\qquad\qquad\qquad\qquad \exists j(rd(P,j,\lambda) \wedge rd(J,j,n) \wedge rd(Q,j,n) \wedge rd(S,n,j))) \wedge \\
\qquad\quad \forall j((1{\leq}j{\leq}l \wedge \neg(j{=}i)) \rightarrow \\
\qquad\qquad ((rd(P,j,ncs) \wedge rd(Q,j,0)) \vee \\
\qquad\qquad\ \exists k(1{\leq}k{<}m \wedge \\
\qquad\qquad\qquad ((rd(P,j,w) \wedge rd(J,j,k{+}1) \wedge rd(Q,j,k)) \vee \\
\qquad\qquad\qquad\ (rd(P,j,\lambda) \wedge rd(J,j,k) \wedge rd(Q,j,k))))))) \wedge \\
\qquad ln(P,l) \ \wedge \ ln(J,l) \ \wedge \ ln(Q,l) \ \wedge \ ln(S,l))
\end{array}
$$

Chapter 8

Appendix C

In the following we provide another example of the program verification technique illustrated in Chapter 5, applied to the parameterized Token Ring protocol. This protocol ensures mutually exclusive access to a transmission medium shared among N processes. A generic process i, with $1 \leq i \leq N$, consists of an infinite loop during which the process tries to obtain access to a shared transmission medium in order to send data. A process can be either in an *idle* state, or in a *try* state, where it is trying to obtain access to the transmission medium, or in a *send* state (see Figure 8.1, where a generic process is represented as a finite state diagram). We assume that every process is initially in its *idle* state.

We want to establish the following *Collsion Avoidance* property of the computation of the given system of N processes: *for all i and j in $\{1, \ldots, N\}$, while process i sends data on the transmission medium, process j, with $j \neq i$, does not send any data.*

Our encoding of the Token Ring protocol makes use of one array $P[0, \ldots, N-1]$ whose elements are in the set $\{idle, try, send\}$, encoding the state of each process, and of a variable T, on the natural numbers, which is shared among the N processes and is equal to the identifier i of the process that currently holds the token. The states $P[0], \ldots, P[N-1]$ of the N processes are initially set to *idle* and the variable T is initially set to $N-1$.

---

while $true$ **do**
  $idle$: **if** $T = i$ **then** $T := T + 1 \bmod N$
    goto $\{idle, try\}$;
  try: **await** $T = i$;
  $send$: **send_data**;
od

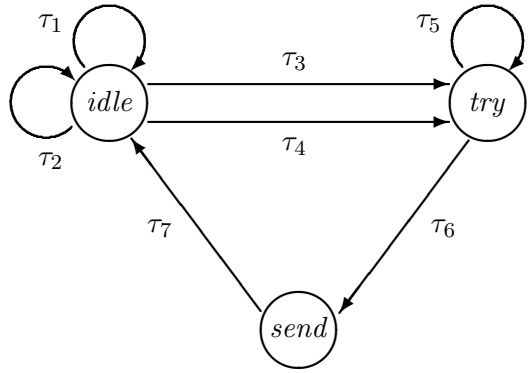

Figure 8.1: Token Ring protocol. (Above) Pseudocode of a generic process i. (Below) The corresponding finite state diagram. Formulas $\tau_1, \ldots, \tau_7$ are defined in the text. *idle* is the initial state.

Notice that for $i = 1, \ldots, N$, process i is nondeterministic in the sense that in some of its states more than one transition is possible. However, we assume that exactly one of the system enabled transitions takes place. Note that we do not make any fairness assumptions.

In the system execution we assume that the sequence of the test $T = i$, the (possible) assignment $T := T+1 \bmod N$ and the jump **goto** $\{idle, try\}$; is atomic. We have made these atomicity assumptions (which correspond to the labels of the transitions of the diagram of Figure 8.1) for keeping the presentation of our proof of the collision avoidance property as simple as possible.

We assume that the number N of processes does *not* change over time, in the sense that, while the computation progresses, neither a new process is constructed nor an existing process is destroyed.

We now present the protocol as a set of transitions between states. In this specification a *state* is represented by a term of the form $s(P, T)$, where: (i) P is an array of the form $[p_0, \ldots, p_{N-1}]$ such that, for $i = 0, \ldots, N-1$, p_i is a constant in the set $\{idle, try, send\}$ representing the state of process $i+1$, and (ii) $T \in \{0, \ldots, N-1\}$.

The transition relation is defined by the seven statements $T_1, \ldots, T_7$ which we now introduce. For $r = 1, \ldots, 7$, statement T_r is of the form:

$$t(s(P, J, Q, S), s(P', J', Q', S')) \leftarrow \tau_r(s(P, J, Q, S), s(P', J', Q', S'))$$

where $\tau_r(s(P, J, Q, S), s(P', J', Q', S'))$ is an array formula defined as follows (see also Figure 8.1):

1. For the first transition from *idle* to *idle*:
   $\tau_1(s(P,T),s(P',T')) \equiv_{def}$
   $\exists i\ (rd(P, i, idle) \wedge T \neq i) \wedge P' = P \wedge T' = T$

2. For the second transition from *idle* to *idle*:
   $\tau_2(s(P,T),s(P',T')) \equiv_{def}$
   $\exists i\ (rd(P, i, idle) \wedge T = i) \wedge P' = P \wedge T' = T+1 \bmod N$

3. For the first transition from *idle* to *try*:
   $\tau_3(s(P,T),s(P',T')) \equiv_{def}$
   $\exists i\ (rd(P, i, idle) \wedge T = i \wedge wr(P, i, try, P')) \wedge T' = T+1 \bmod N$

4. For the second transition from *idle* to *try*:
   $\tau_4(s(P,T),s(P',T')) \equiv_{def}$
   $\exists i\ (rd(P, i, idle) \wedge T \neq i \wedge wr(P, i, try, P')) \wedge T' = T$

5. For the transition from *try* to *try*:
   $\tau_5(s(P,T),s(P',T')) \equiv_{def}$
   $\exists i\ (rd(P, i, try) \wedge T \neq i) \wedge P' = P \wedge T' = T$

6. For the transition from *try* to *send*:
$\tau_6(s(P,T),s(P',T')) \equiv_{def}$
$\exists i\ (rd(P,i,try) \wedge T=i \wedge wr(P,i,send,P')) \wedge T'=T$

7. For the transition from *send* to *idle*:
$\tau_7(s(P,T),s(P',T')) \equiv_{def}$
$\exists i\ (rd(P,i,send) \wedge wr(P,i,idle,P')) \wedge T'=T$

Following the approach of Chapter 5, the program P_{TR} is the specification of the parameterized Token Ring protocol. P_{TR} is constructed as illustrated in Chapter 5 except for the transition relation τ that we have defined above and the following definitions of the *initial* and *collision* atomic properties:

A_1: $atomic(s(P,T), initial) \leftarrow$
$\forall i\,(0 \leq i \leq N-1 \rightarrow rd(P,i,idle)) \wedge T = N-1 \bmod N$
A_2: $atomic(s(P,T), collision) \leftarrow$
$\exists i,j\ (rd(P,i,send)\ \wedge\ rd(P,j,send)\ \wedge \neg(j=i))$

The collision avoidance property of the Token Ring protocol is expressed by the following temporal formula:

φ: $initial \rightarrow \neg\, EF\ collision$

In particular, we now prove that:

$M(P_{TR}) \models \forall X\ holds(X, initial \rightarrow \neg\, ef(collision))$

In order to verify the Collision Avoidance property of the Token Ring protocol we start from the statement:

$nocollision \leftarrow \forall X\ holds(X, initial \rightarrow \neg\, ef(collision))$

and, by applying the Lloyd-Topor transformation, we get the following two clauses:

1. $nocollision \leftarrow \neg\, new1$
2. $new1 \leftarrow holds(X, initial) \wedge holds(X, ef(collision))$

Now we apply the transformation strategy illustrated in Chapter 5 starting from $P_{TR} \cup \{1,2\}$. By unfolding clause 2 w.r.t. the second atom, we get:

3. $new1 \leftarrow atomic(s(P,T), initial) \wedge \tau_1(s(P,T), s(P',T')) \wedge$
   $holds(s(P',T'), ef(collision))$
4. $new1 \leftarrow atomic(s(P,T), initial) \wedge \tau_2(s(P,T), s(P',T')) \wedge$
   $holds(s(P',T'), ef(collision))$
5. $new1 \leftarrow atomic(s(P,T), initial) \wedge \tau_3(s(P,T), s(P',T')) \wedge$
   $holds(s(P',T'), ef(collision))$
6. $new1 \leftarrow atomic(s(P,T), initial) \wedge \tau_4(s(P,T), s(P',T')) \wedge$
   $holds(s(P',T'), ef(collision))$

where we have removed the clauses whose bodies contain the following goals

$$atomic(s(P,T), initial) \wedge atomic(s(P,T), collision)$$

and

$$atomic(s(P,T), initial) \wedge \tau_i(s(P,T), s(P',T')) \qquad (\text{for } i \in \{5,6,7\})$$

because the corresponding array formulas are unsatisfiable. By definition introduction we add the following clauses to the current program:

7. $new2 \leftarrow genc_1(s(P,T)) \wedge holds(s(P,T), ef(collision))$
8. $new3 \leftarrow genc_2(s(P,T)) \wedge holds(s(P,T), ef(collision))$
9. $new4 \leftarrow genc_3(s(P,T)) \wedge holds(s(P,T), ef(collision))$

where the array formulas $genc_1$, $genc_2$, and $genc_3$ have been computed by generalization from the array formulas occurring in the body of the clauses 4,5, and 6, respectively. Then, by folding clauses 3,4,5, and 6 we obtain:

3'. $new1 \leftarrow new1$
4'. $new1 \leftarrow atomic(s(P,T), initial) \wedge \tau_2(s(P,T), s(P',T')) \wedge new2(s(P',T'))$
5'. $new1 \leftarrow atomic(s(P,T), initial) \wedge \tau_3(s(P,T), s(P',T')) \wedge new3(s(P',T'))$
6'. $new1 \leftarrow atomic(s(P,T), initial) \wedge \tau_4(s(P,T), s(P',T')) \wedge new4(s(P',T'))$

Now the transformation strategy continues by transforming clauses 7,8, and 9 until no new definitions are introduced. The transformation process of phase A terminates and returns the following program.

1. $nocollision \leftarrow \neg\, new1$ Program TR

3'. $new1 \leftarrow new1$
4'. $new1 \leftarrow atomic(a, initial) \wedge \tau_2(a, a') \wedge new2(a')$
5'. $new1 \leftarrow atomic(a, initial) \wedge \tau_3(a, a') \wedge new3(a')$
6'. $new1 \leftarrow atomic(a, initial) \wedge \tau_4(a, a') \wedge new4(a')$

10. $new2(a) \leftarrow genc_1(a) \wedge \tau_1(a, a') \wedge new2(a')$
11. $new2(a) \leftarrow genc_1(a) \wedge \tau_2(a, a') \wedge new2(a')$
12. $new2(a) \leftarrow genc_1(a) \wedge \tau_3(a, a') \wedge new3(a')$
13. $new2(a) \leftarrow genc_1(a) \wedge \tau_4(a, a') \wedge new4(a')$

14. $new3(a) \leftarrow genc_2(a) \wedge \tau_1(a, a') \wedge new3(a')$
15. $new3(a) \leftarrow genc_2(a) \wedge \tau_2(a, a') \wedge new5(a')$
16. $new3(a) \leftarrow genc_2(a) \wedge \tau_3(a, a') \wedge new6(a')$
17. $new3(a) \leftarrow genc_2(a) \wedge \tau_4(a, a') \wedge new6(a')$
18. $new3(a) \leftarrow genc_2(a) \wedge \tau_5(a, a') \wedge new3(a')$

19. $new4(a) \leftarrow genc_3(a) \wedge \tau_1(a, a') \wedge new4(a')$
20. $new4(a) \leftarrow genc_3(a) \wedge \tau_2(a, a') \wedge new5(a')$
21. $new4(a) \leftarrow genc_3(a) \wedge \tau_3(a, a') \wedge new6(a')$
22. $new4(a) \leftarrow genc_3(a) \wedge \tau_4(a, a') \wedge new7(a')$
23. $new4(a) \leftarrow genc_3(a) \wedge \tau_5(a, a') \wedge new4(a')$

24. $new5(a) \leftarrow genc_4(a) \wedge \tau_1(a, a') \wedge new5(a')$
25. $new5(a) \leftarrow genc_4(a) \wedge \tau_2(a, a') \wedge new5(a')$
26. $new5(a) \leftarrow genc_4(a) \wedge \tau_3(a, a') \wedge new6(a')$
27. $new5(a) \leftarrow genc_4(a) \wedge \tau_4(a, a') \wedge new8(a')$
28. $new5(a) \leftarrow genc_4(a) \wedge \tau_5(a, a') \wedge new7(a')$
29. $new5(a) \leftarrow genc_4(a) \wedge \tau_6(a, a') \wedge new9(a')$

30. $new6(a) \leftarrow genc_5(a) \wedge \tau_1(a, a') \wedge new6(a')$
31. $new6(a) \leftarrow genc_5(a) \wedge \tau_2(a, a') \wedge new8(a')$
32. $new6(a) \leftarrow genc_5(a) \wedge \tau_3(a, a') \wedge new6(a')$
33. $new6(a) \leftarrow genc_5(a) \wedge \tau_4(a, a') \wedge new6(a')$
34. $new6(a) \leftarrow genc_5(a) \wedge \tau_5(a, a') \wedge new6(a')$
35. $new6(a) \leftarrow genc_5(a) \wedge \tau_6(a, a') \wedge new10(a')$

36. $new7(a) \leftarrow genc_6(a) \wedge \tau_1(a, a') \wedge new7(a')$
37. $new7(a) \leftarrow genc_6(a) \wedge \tau_2(a, a') \wedge new8(a')$
38. $new7(a) \leftarrow genc_6(a) \wedge \tau_3(a, a') \wedge new8(a')$
39. $new7(a) \leftarrow genc_6(a) \wedge \tau_4(a, a') \wedge new7(a')$
40. $new7(a) \leftarrow genc_6(a) \wedge \tau_5(a, a') \wedge new7(a')$
41. $new8(a) \leftarrow genc_7(a) \wedge \tau_1(a, a') \wedge new8(a')$
42. $new8(a) \leftarrow genc_7(a) \wedge \tau_2(a, a') \wedge new8(a')$
43. $new8(a) \leftarrow genc_7(a) \wedge \tau_3(a, a') \wedge new8(a')$
44. $new8(a) \leftarrow genc_7(a) \wedge \tau_4(a, a') \wedge new8(a')$
45. $new8(a) \leftarrow genc_7(a) \wedge \tau_5(a, a') \wedge new8(a')$
46. $new8(a) \leftarrow genc_7(a) \wedge \tau_6(a, a') \wedge new10(a')$
47. $new9(a) \leftarrow genc_8(a) \wedge \tau_1(a, a') \wedge new9(a')$
48. $new9(a) \leftarrow genc_8(a) \wedge \tau_4(a, a') \wedge new10(a')$
49. $new9(a) \leftarrow genc_8(a) \wedge \tau_7(a, a') \wedge new2(a')$
50. $new10(a) \leftarrow genc_9(a) \wedge \tau_1(a, a') \wedge new10(a')$
51. $new10(a) \leftarrow genc_9(a) \wedge \tau_4(a, a') \wedge new10(a')$
52. $new10(a) \leftarrow genc_9(a) \wedge \tau_5(a, a') \wedge new10(a')$
53. $new10(a) \leftarrow genc_9(a) \wedge \tau_7(a, a') \wedge new8(a')$

Now we proceed to Phase B of our strategy. Since in program TR the predicates $new1$ through $new10$ are useless, we remove clauses 3',4',5',6', and clauses 10 through 53. By doing so, we derive a program consisting of clause 1 only. By unfolding clause 1 we get the final program T, which consists of the clause $nocollision \leftarrow$ only. Thus, $M(T) \models nocollision$ and we have proved that:

$M(P_{TR}) \models \forall X \; holds(X, initial \rightarrow \neg \, ef(nocollision))$

As a consequence, we have that for any initial state and for any number $N(\geq 2)$ of processes, the Token Ring protocol ensures an exclusive access to the transmission medium. In the following we give the definitions of the array formulas $genc_1$ through $genc_8$ occurring in the program TR.

$genc_1(s(P, T)) \equiv_{def}$
$\quad \exists i \, (T = i \bmod N) \wedge \forall i \, (0 \leq i \wedge i \leq N-1 \rightarrow rd(P, i, idle))$

$$genc_2(s(P,T)) \equiv_{def}$$
$$\exists j\,(0 \le j \wedge j \le N{-}1 \wedge rd(P,j,try) \wedge T = j{+}1 \bmod N \wedge$$
$$\forall i\,(0 \le i \wedge i \le N{-}1 \wedge i \ne j \rightarrow rd(P,i,idle)))$$

$$genc_3(s(P,T)) \equiv_{def}$$
$$\exists j,m\,(0 \le j \wedge j \le N{-}1 \wedge rd(P,j,try) \wedge T \ne j \wedge T = m \bmod N \wedge$$
$$\forall i\,(0 \le i \wedge i \le N{-}1 \wedge i \ne j \rightarrow rd(P,i,idle)))$$

$$genc_4(s(P,T)) \equiv_{def}$$
$$\exists j,m\,(0 \le j \wedge j \le N{-}1 \wedge rd(P,j,try) \wedge T = m \bmod N \wedge$$
$$\forall i\,(0 \le i \wedge i \le N{-}1 \wedge i \ne j \rightarrow rd(P,i,idle)))$$

$$genc_5(s(P,T)) \equiv_{def}$$
$$\exists j\,(0 \le j \wedge j \le N{-}1 \wedge rd(P,j,try) \wedge T = j{+}1 \bmod N \wedge$$
$$\forall i\,(0 \le i \wedge i \le N{-}1 \wedge i \ne j \rightarrow (rd(P,i,idle) \vee rd(P,i,try))))$$

$$genc_6(s(P,T)) \equiv_{def}$$
$$\exists j\,(0 \le j \wedge j \le N{-}1 \wedge T = j \bmod N \wedge$$
$$\forall i\,(0 \le i \wedge i \le N{-}1 \rightarrow (rd(P,i,idle) \vee (rd(P,i,try) \wedge i \ne j))))$$

$$genc_7(s(P,T)) \equiv_{def}$$
$$\exists j\,(T = j \bmod N \wedge$$
$$\forall i\,(0 \le i \wedge i \le N{-}1 \rightarrow (rd(P,i,idle) \vee rd(P,i,try))))$$

$$genc_8(s(P,T)) \equiv_{def}$$
$$\exists j\,(0 \le j \wedge j \le N{-}1 \wedge rd(P,j,send) \wedge T = j \bmod N \wedge$$
$$\forall i\,(0 \le i \wedge i \le N{-}1 \wedge i \ne j \rightarrow rd(P,i,idle)))$$

$$genc_9(s(P,T)) \equiv_{def}$$
$$\exists j\,(0 \le j \wedge j \le N{-}1 \wedge rd(P,j,send) \wedge T = j \bmod N \wedge$$
$$\forall i\,(0 \le i \wedge i \le N{-}1 \wedge i \ne j \rightarrow (rd(P,i,idle) \vee rd(P,i,try))))$$

Bibliography

[1] K. R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 493–576. Elsevier, 1990.

[2] K. R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[3] K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19, 20:9–71, 1994.

[4] K. R. Apt and D. C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, 1986.

[5] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. D. Zuck. Parameterized verification with automatically computed inductive assertions. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV '01)*, pages 221–234. Springer, 2001.

[6] F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.

[7] Franz Baader and Cesare Tinelli. Deciding the word problem in the union of equational theories sharing constructors. In P. Narendran and M. Rusinowitch, editors, *Proceedings of the 10th International*

Conference on Rewriting Techniques and Applications (Trento, Italy), volume 1631, pages 175–189. Springer-Verlag, 1999.

[8] Dan Benanav, Deepak Kapur, and Paliath Narendran. Complexity of matching problems. *J. Symb. Comput.*, 3(1-2):203–216, 1987.

[9] N. Bensaou and I. Guessarian. Transforming constraint logic programs. *Theoretical Computer Science*, 206:81–125, 1998.

[10] M. Bezem. Characterizing termination of logic programs with level mappings. In E.L. Lusk and R.A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming, Cleveland, Ohio (USA)*, pages 69–80. MIT Press, 1989.

[11] A. Bik and H. Wijshoff. Implementation of Fourier-Motzkin elimination, 1994.

[12] A. Bossi and N. Cocco. Preserving universal termination through unfold/fold. In *Proceedings ALP '94*, Lecture Notes in Computer Science 850, pages 269–286, Berlin, 1994. Springer-Verlag.

[13] A. Bossi, N. Cocco, and S. Etalle. On safe folding. In *Proceedings PLILP '92, Leuven, Belgium*, Lecture Notes in Computer Science 631, pages 172–186. Springer-Verlag, 1992.

[14] A. Bundy. The automation of proof by mathematical induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, pages 845–911. North Holland, 2001.

[15] H.-J. Bürckert. Matching—a special case of unification? *J. Symb. Comput.*, 8(5):523–536, 1989.

[16] Hans-Jürgen Bürckert. Some relationships between unification, restricted unification, and matching. In *Proceedings of the 8th International Conference on Automated Deduction*, pages 514–524, London, UK, 1986. Springer-Verlag.

[17] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[18] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[19] J. Cook and J. P. Gallagher. A transformation system for definite programs based on termination analysis. In L. Fribourg and F. Turini, editors, *Proceedings of LoPSTr'94 and META'94, Pisa, Italy*, Lecture Notes in Computer Science 883, pages 51–68. Springer-Verlag, 1994.

[20] B. Courcelle. Equivalences and transformations of regular systems – applications to recursive program schemes and grammars. *Theoretical Computer Science*, 42:1–122, 1986.

[21] G. Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*, 23(3):257–301, 2003.

[22] G. Delzanno and A. Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, 2001.

[23] S. M. Eker. Improving the efficiency of AC matching and unification. Technical Report RR-2104, Institut de Recherche en Informatique et en Automatique, Lorraine, 1993.

[24] S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.

[25] F. Fioravanti, A. Pettorossi, and M. Proietti. Automated strategies for specializing constraint logic programs. In K.-K. Lau, editor, *Proceedings of LOPSTR 2000, Tenth International Workshop on Logic-based Program Synthesis and Transformation, London, UK, 24-28 July, 2000*, Lecture Notes in Computer Science 2042, pages 125–146. Springer-Verlag, 2001.

[26] F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. In *Proceedings of the ACM Sigplan Workshop on Verification and Computational Logic VCL'01, Florence (Italy)*, Technical Report DSSE-TR-2001-3, pages 85–96. University of Southampton, UK, 2001.

[27] F. Fioravanti, A. Pettorossi, and M. Proietti. Specialization with clause splitting for deriving deterministic constraint logic programs. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Hammamet (Tunisia)*. IEEE Computer Society Press, 2002.

[28] F. Fioravanti, A. Pettorossi, and M. Proietti. Verification of sets of infinite state systems using program transformation. In A. Pettorossi, editor, *Proceedings of LOPSTR 2001, Eleventh International Workshop on Logic-based Program Synthesis and Transformation*, Lecture Notes in Computer Science 2372, pages 111–128. Springer-Verlag, 2002.

[29] F. Fioravanti, A. Pettorossi, and M. Proietti. Transformation rules for locally stratified constraint logic programs. In K.-K. Lau and M. Bruynooghe, editors, *Program Development in Computational Logic*, Lecture Notes in Computer Science 3049, pages 292–340. Springer, 2004.

[30] L. Fribourg and H. Olsén. A decompositional approach for computing least fixed-points of Datalog programs with Z-counters. *Constraints*, 2(3/4):305–335, 1997.

[31] M. Gergatsoulis and M. Katzouraki. Unfold/fold transformations for definite clause programs. In M. Hermenegildo and J. Penjam, editors, *Proceedings Sixth International Symposium on Programming Language Implementation and Logic Programming (PLILP '94)*, Lecture Notes in Computer Science 844, pages 340–354. Springer-Verlag, 1994.

[32] J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[33] T. Kanamori and H. Fujita. Unfold/fold transformation of logic programs with counters. Technical Report 179, ICOT, Tokyo, Japan, 1986.

[34] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, December 1984.

[35] L. Kott. About transformation system: A theoretical study. In *3ème Colloque International sur la Programmation*, pages 232–247, Paris (France), 1978. Dunod.

[36] L. Kott. The McCarthy's induction principle: 'oldy' but 'goody'. *Calcolo*, 19(1):59–69, 1982.

[37] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[38] K.-K. Lau, M. Ornaghi, A. Pettorossi, and M. Proietti. Correctness of logic program transformation based on existential termination. In J. W. Lloyd, editor, *Proceedings of the 1995 International Logic Programming Symposium (ILPS '95)*, pages 480–494. MIT Press, 1995.

[39] R. Lazic, T. C. Newcomb, and A. W. Roscoe. On model checking data-independent systems with arrays with whole-array operations. In *Communicating Sequential Processes: The First 25 Years*, Lecture Notes in Computer Science 3525, pages 275–291. Springer, 2004.

[40] M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4&5):461–515, 2002.

[41] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialization. In A. Bossi, editor, *Proceedings of LOPSTR '99, Venice, Italy*, Lecture Notes in Computer Science 1817, pages 63–82. Springer, 1999.

[42] M. Livesey and J. Siekmann. Unification of A+C Terms (Bags) and A+C+I Terms (Sets). Technical Report Interner Bericht Nr. 3/76, Institut for informatik I, Universitt Karlsruhe, 1976.

[43] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second Edition.

[44] Rüdiger Loos and Volker Weispfenning. Applying linear quantifier elimination. *The Computer Journal*, 36(5):450–462, 1993.

[45] M. J. Maher. A transformation system for deductive database modules with perfect model semantics. *Theoretical Computer Science*, 110:377–403, 1993.

[46] MAP group. The MAP transformation system. Available from `http://www.iasi.rm.cnr.it/~proietti/system.html`, 1995–2008.

[47] J. McCarthy. Towards a mathematical science of computation. In C.M. Popplewell, editor, *Information Processing* : *Proceedings of IFIP 1962*, pages 21–28, Amsterdam, 1963. North Holland.

[48] K. L. McMillan, S. Qadeer, and J. B. Saxe. Induction in compositional model checking. In *CAV 2000*, Lecture Notes in Computer Science 1855, pages 312–327. Springer, 2000.

[49] E. Mendelson. *Introduction to Mathematical Logic*. Wadsworth & Brooks/Cole Advanced Books & Software, Monterey, California, Usa, Monterey, California, Usa, 1987. Third Edition.

[50] U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model-checking. In J. W. Lloyd et al., editor, *First International Conference on Computational Logic, CL 2000, London, UK, 24-28 July, 2000*, Lecture Notes in Artificial Intelligence 1861, pages 384–398. Springer-Verlag, 2000.

[51] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[52] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.

[53] A. Pettorossi and M. Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):360–414, 1996.

[54] A. Pettorossi and M. Proietti. Synthesis and transformation of logic programs using unfold/fold proofs. *Journal of Logic Programming*, 41(2&3):197–230, 1999.

[55] A. Pettorossi and M. Proietti. Perfect model checking via unfold/fold transformations. In J. W. Lloyd, editor, *First International Conference on Computational Logic, CL 2000, London, UK, 24-28 July, 2000*, Lecture Notes in Artificial Intelligence 1861, pages 613–628. Springer, 2000.

[56] A. Pettorossi and M. Proietti. A theory of totally correct logic program transformations. In *2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 2004, August 24-25, 2004, Verona, Italy*, pages 159–168. ACM Press, 2004.

[57] A. Pettorossi, M. Proietti, and V. Senni. Proving properties of constraint logic programs by eliminating existential variables. In S. Etalle and M. Truszczynski, editors, *Proceedings of the 22nd International Conference on Logic Programming (ICLP '06)*, volume 4079 of *Lecture Notes in Computer Science*, pages 179–195. Springer-Verlag, 2006.

[58] A. Pettorossi, M. Proietti, and V. Senni. Transformational verification of parameterized protocols using array formulas. In P.M. Hill, editor, *Proceedings of the 15th International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR '05)*, volume 3901 of *Lecture Notes in Computer Science*, pages 23–43. Springer-Verlag, 2006.

[59] A. Pettorossi, M. Proietti, and V. Senni. Automatic correctness proofs for logic program transformations. In V. Dahl and I. Niemelä, editors, *Proceedings of the 23rd International Conference on Logic Programming (ICLP '07)*, pages 364–379, 2007.

[60] M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. *Theoretical Computer Science*, 142(1):89–124, 1995.

[61] T. C. Przymusinski. On the declarative semantics of stratified deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1988.

[62] M. O. Rabin. Decidable theories. In Jon Barwise, editor, *Handbook of Mathematical Logic*, pages 595–629. North-Holland, 1977.

[63] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *CAV '97*, Lecture Notes in Computer Science 1254, pages 143–154. Springer-Verlag, 1997.

[64] Christophe Ringeissen. Matching in a class of combined non-disjoint theories. In Franz Baader, editor, *19th International Conference on Automated Deduction - CADE'2003, Miami, Floride, USA*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 212–227. Geoff Sutcliffe, Springer-Verlag, Aug 2003.

[65] A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, and I. V. Ramakrishnan. Beyond Tamaki-Sato style unfold/fold transformations for normal logic programs. *International Journal on Foundations of Computer Science*, 13(3):387–403, 2002.

[66] A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, and I. V. Ramakrishnan. An unfold/fold transformation framework for definite logic programs. *ACM Transactions on Programming Languages and Systems*, 26:264–509, 2004.

[67] A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Verification of parameterized systems using logic program transformations. In *Proceedings of the Sixth International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000, Berlin, Germany*, Lecture Notes in Computer Science 1785, pages 172–187. Springer, 2000.

[68] A. Roychoudhury and C. R. Ramakrishnan. Unfold/fold transformations for automated verification. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*, Lecture Notes in Computer Science 3049, pages 261–290. Springer, 2004.

[69] A. Roychoudhury and I. V. Ramakrishnan. Automated inductive verification of parameterized protocols. In *CAV 2001*, pages 25–37, 2001.

[70] D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Toplas*, 18(2):175–234, 1996.

[71] T. Sato and H. Tamaki. Examples of logic program transformation and synthesis. Case studies of transformation and synthesis of logic programs done up to Feb. 85, 1985.

[72] Manfred Schmidt-Schaub. Unification in a combination of arbitrary disjoint equational theories. *J. Symb. Comput.*, 8(1-2):51–99, 1989.

[73] H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.

[74] V. Senni. Transformational verification of the parameterized Peterson's protocol. Available from the author, Unpublished note, July 2005.

[75] V. Senni, A. Pettorossi, and M. Proietti. Folding transformation rules for constraint logic programs. Submitted for publication, April 2008.

[76] N. Shankar. Combining theorem proving and model checking through symbolic analysis. In *CONCUR 2000: Concurrency Theory*, number

1877 in Lecture Notes in Computer Science, pages 1–16, State College, PA, August 2000. Springer-Verlag.

[77] Mark E. Stickel. A unification algorithm for associative-commutative functions. *J. ACM*, 28(3):423–434, 1981.

[78] A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for an extensional theory of arrays. In *16th IEEE Symposium on Logic in Computer Science*, pages 29–37. IEEE Press, 2001.

[79] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, editor, *Proceedings of the Second International Conference on Logic Programming*, pages 127–138, Uppsala, Sweden, 1984. Uppsala University.

[80] H. Tamaki and T. Sato. A generalized correctness proof of the unfold/fold logic program transformation. Technical Report 86-4, Ibaraki University, Japan, 1986.

[81] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

[82] Volker Weispfenning. The complexity of linear problems in fields. *J. Symb. Comput.*, 5(1-2):3–27, 1988.

[83] L. D. Zuck and A. Pnueli. Model checking and abstraction to the aid of parameterized systems (a survey). *Computer Languages, Systems & Structures*, 30(3-4):139–169, 2004.