Selective Unification in Constraint Logic Programming

Frédéric Mesnard, Etienne Payet, Germán Vidal

Selective Unification in Constraint Logic Programming extended version with proofs *

Fred Mesnard[1], Étienne Payet[1], and Germán Vidal[2]

[1] LIM - Université de la Réunion, France
{frederic.mesnard,etienne.payet}@univ-reunion.fr
[2] MiST, DSIC, Universitat Politècnica de València, Spain
gvidal@dsic.upv.es

Abstract. Concolic testing is a well-known validation technique for imperative and object-oriented programs. We have recently introduced an adaptation of this technique to logic programming. At the heart of our framework for concolic testing lies a logic programming specific procedure that we call "selective unification". In this paper, we consider concolic testing in the context of constraint logic programming and extend the notion of selective unification accordingly. We prove that the selective unification problem is generally undecidable for constraint logic programs, and we present a correct and complete algorithm for selective unification in the context of a class of constraint structures.

1 Introduction

Concolic testing is a well-known validation technique for imperative and object-oriented programs. Roughly speaking, concolic testing combines *conc*rete and symb*olic* execution (called *concolic* execution) in order to systematically produce test cases that aim at exploring all feasible execution paths of a program. Typically, one starts with an arbitrary concrete call, say $main(i_1, \ldots, i_n)$ and runs both concrete and symbolic execution on $main(i_1, \ldots, i_n)$ and $main(v_1, \ldots, v_n)$, respectively, where $v_1, \ldots, v_n$ are symbolic variables denoting unknown input values. In contrast to ordinary symbolic execution, the symbolic component of concolic execution does not explore all possible execution paths, but just mimics the steps of the concrete execution, gathering along the way constraints on the symbolic variables to follow a particular path. Once a concolic execution terminates (and it usually terminates, since concrete executions are assumed terminating), one uses the collected constraints to produce new concrete calls that will explore different paths. E.g., if the collected constraints are c_1, c_2, c_3, then by solving $\neg c_1$ we get values for the symbolic variables of $main(v_1, \ldots, v_n)$ so that the resulting concrete call will follow a different execution path. Other alternative initial calls can be obtained by solving $c_1 \wedge \neg c_2$ and $c_1 \wedge c_2 \wedge \neg c_3$.

We have recently introduced an adaptation of this technique to logic programming [16, 17]. In contrast to the case of imperative or object-oriented programming, computing the alternatives of a given execution is more complex in logic programming. Consider, e.g., a predicate p/n defined by the set of clauses

$$\{H_1 \leftarrow B_1, H_2 \leftarrow B_2, H_3 \leftarrow B_3\}$$

* This work has been partially supported by MINECO/AEI/FEDER (EU) under grants TIN2013-44742-C4-1-R and TIN2016-76843-C4-1-R, and by the *Generalitat Valenciana* under grant PROMETEO-II/2015/013 (SmartLogic).

and a goal where the selected atom, $p(t_1, \ldots, t_n)$, only unifies with H_1. What are then the possible alternatives? In principle, one could think that producing a goal where the selected atom only unifies with H_2 and another goal where the selected atom only unifies with H_3 is enough. However, there are five more possibilities: unifiying with no clause, unifiying with both H_1 and H_2, unifying with both H_1 and H_3, unifying with both H_2 and H_3, and unifying with all three atoms H_1, H_2 and H_3.[3] Moreover, we found in [16, 17] that producing goals that satisfy each of these conditions is far from trivial. This problem, called "selective unification" in [17], can be roughly expressed as follows: given an atom A, a set of *positive atoms* $\mathcal{H}^+$ and a set of *negative atoms* $\mathcal{H}^-$, we look for a substitution θ (if it exists) for the variables of A such that $A\theta$ unifies with every atom in $\mathcal{H}^+$ but it does not unify with any atom in $\mathcal{H}^-$. Observe that we want $A\theta$ to unify with each atom in $\mathcal{H}^+$ separately.

Let us illustrate the notion of selective unification with a simple example. Consider, e.g., an atom $p(X)$ and the sets $\mathcal{H}^+ = \{p(f(a)), p(f(b))\}$ and $\mathcal{H}^- = \{p(c)\}$. A solution of this selective unification problem is $\{X/f(Y)\}$, since $p(X)\{X/f(Y)\} = p(f(Y))$ unifies with $p(f(a))$, it also unifies with $p(f(b))$ (with different unifiers, though), but it does not unify with $p(c)$. In contrast, the problem with atom $p(X)$ and sets $\mathcal{H}^+ = \{p(a), p(b)\}$ and $\mathcal{H}^- = \{p(c)\}$ is unfeasible since we need a renaming, e.g., $\theta = \{X/Y\}$ in order for $p(X)\theta$ to unify with both $p(a)$ and $p(b)$, but then $p(X)\theta$ would also unify with $p(c)$.

In [16, 17] we also considered a *groundness* condition that requires some arguments of the initial goal to be ground. This condition is required to ensure that the produced goals are valid *run time* goals (thus ensuring they are appropriate test cases), see Definition 1 in Section 4.1. Although the problem is decidable in some cases, finding an efficient algorithm is rather complex (see [17] for more details).

In this paper, we consider concolic testing in the setting of constraint logic programming (CLP) [9, 10]. We prove that the selective unification problem is generally undecidable for CLP programs, and we present a correct and complete algorithm for selective unification in the context of a class of constraint structures.

The paper is organized as follows. In Section 2, we present the main definitions about CLP. In Section 3, we recall the framework for concolic testing for LP and sketch its generalization to CLP. Section 4 focuses on *selective unification*. In Section 5, we show that selective unification is in general undecidable for CLP. In Section 6, we add assumptions on constraint structures which help solving selective unification problems. Finally, Section 7 discusses related work and concludes the paper.

2 Preliminary Definitions

The set of natural numbers is denoted by $\mathbb{N}$. We recall some basic definitions about CLP, see [10, 11] for more details. From now on, we fix an infinite countable set $\mathcal{V}$ of *variables* together with a *signature* Σ, i.e., a pair $\langle F, \Pi_C \rangle$ where F is a finite set of *function symbols* and Π_C is a finite set of *predicate symbols* with $F \cap \Pi_C = \emptyset$ and $(F \cup \Pi_C) \cap \mathcal{V} = \emptyset$. Every element of $F \cup \Pi_C$ has an *arity* which is the number of its arguments. We write $f/n \in F$ (resp. $p/n \in \Pi_C$) to denote that f (resp. p) is an element of F (resp. Π_C) whose arity is $n \geq 0$. A *constant symbol* is an element of F whose arity is 0.

A *term* is a variable, a constant symbol or an entity $f(t_1, \ldots, t_n)$ where $f/n \in F$, $n \geq 1$ and $t_1, \ldots, t_n$ are terms. A term is *ground* when it contains no variable. An *atomic constraint* is an element $p/0$ of Π_C or an entity $p(t_1, \ldots, t_n)$ where $p/n \in \Pi_C$, $n \geq 1$ and $t_1, \ldots, t_n$ are terms. A first-order *formula* on Σ is built from atomic constraints in the usual way using the logical connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$ and the quantifiers $\exists$ and $\forall$. Let ϕ be a formula. Then $\mathcal{V}ar(\phi)$ denotes its set of free variables. We let $\exists\phi$ (resp. $\forall\phi$) denote the existential (resp. universal) closure of ϕ.

[3] In general, though, not all possibilities are feasible.

We fix a Σ-*structure* $\mathcal{D}$, i.e., a pair $\langle D, [\cdot]\rangle$ which is an interpretation of the symbols in Σ. The set D is called the *domain* of $\mathcal{D}$ and $[\cdot]$ maps each $f/0 \in F$ to an element of D and each $f/n \in F$ with $n \geq 1$ to a function $[f] : D^n \to D$; each $p/0 \in \Pi_C$ to an element of $\{0,1\}$ and each $p/n \in \Pi_C$ with $n \geq 1$ to a boolean function $[p] : D^n \to \{0,1\}$. We assume that the binary predicate symbol $=$ is in Σ and is interpreted as identity in D. A *valuation* is a mapping from $\mathcal{V}$ to D. Each valuation v extends by morphism to terms. As usual, a valuation v induces a valuation $[\cdot]_v$ of terms to D and of formulas to $\{0,1\}$.

Given a formula ϕ and a valuation v, we write $\mathcal{D} \models_v \phi$ when $[\phi]_v = 1$. We write $\mathcal{D} \models \phi$ when $\mathcal{D} \models_v \phi$ for all valuation v. Notice that $\mathcal{D} \models \forall\phi$ if and only if $\mathcal{D} \models \phi$, that $\mathcal{D} \models \exists\phi$ if and only if there exists a valuation v such that $\mathcal{D} \models_v \phi$, and that $\mathcal{D} \models \neg\exists\phi$ if and only if $\mathcal{D} \models \neg\phi$. We say that a formula ϕ is *satisfiable* (resp. *unsatisfiable*) in $\mathcal{D}$ when $\mathcal{D} \models \exists\phi$ (resp. $\mathcal{D} \models \neg\phi$).

We fix a set $\mathcal{L}$ of admitted formulas, the elements of which are called *constraints*. We suppose that $\mathcal{L}$ is closed under variable renaming, existential quantification and conjunction and that it contains all the atomic constraints. We assume that there is a computable function *solv* which maps each $c \in \mathcal{L}$ to one of `true` or `false` indicating whether c is satisfiable or unsatisfiable in $\mathcal{D}$. We call *solv* the *constraint solver*. A structure $\mathcal{D}$ admits *quantifier elimination* if for each first-order formula ϕ there exists a quantifier-free formula ψ such that $\mathcal{D} \models \forall[\phi \leftrightarrow \psi]$.

Example 1 (CLP($\mathcal{Q}_{lin}$)). The constraint domain $\mathcal{Q}_{lin}$ has $<, \leq, =, \geq, >$ as predicate symbols, $+, -, *$, $/$ as function symbols and sequences of digits as constant symbols. $\mathcal{L}$ is the set of conjunctions of linear atomic constraints. The domain of computation is the structure with the set of rationals, denoted by $\mathbb{Q}$, as domain and where the predicate symbols and the function symbols are interpreted as the usual relations and functions over the rationals. A constraint solver for $\mathcal{Q}_{lin}$ always returning either `true` or `false` is described in [18]. $\mathcal{Q}_{lin}$ admits variable elimination via the Fourier-Motzkin algorithm (see e.g., [15]).

Example 2 (CLP($\mathcal{A}$)). A constraint domain with natural numbers and arrays is presented in Section 5.

Sequences of distinct variables are denoted by $\vec{X}$, $\vec{Y}$ or $\vec{Z}$ and are sometimes considered as sets of variables: we may write $\forall_{\vec{X}}$, $\exists_{\vec{X}}$ or $\vec{X} \cup \vec{Y}$. Sequences of (not necessarily distinct) terms are denoted by $\vec{s}$, $\vec{t}$ or $\vec{u}$. Given two sequences of n terms $\vec{s} := (s_1, \ldots, s_n)$ and $\vec{t} := (t_1, \ldots, t_n)$, we write $\vec{s} = \vec{t}$ to denote the constraint $s_1 = t_1 \wedge \cdots \wedge s_n = t_n$.

The signature in which all programs and queries under consideration are included is $\Sigma_L := \langle F, \Pi_C \cup \Pi_P\rangle$ where Π_P is the set of predicate symbols that can be defined in programs, with $\Pi_C \cap \Pi_P = \emptyset$.

An *atom* has the form $p(t_1, \ldots, t_n)$ where $p/n \in \Pi_P$ and $t_1, \ldots, t_n$ are terms. Given an atom $A := p(\vec{t})$, we write $rel(A)$ to denote the predicate symbol p. A *rule* has the form $H \leftarrow c \wedge \vec{B}$ where H is an atom, c is a satisfiable constraint, and $\vec{B}$ is a finite sequence of atoms. A *program* is a finite set of rules. A *state* has the form $\langle d \,|\, \vec{B}\rangle$ where $\vec{B}$ is a sequence of atoms and d is a satisfiable constraint. A *constraint atom* is a state of the form $\langle d \,|\, p(\vec{t})\rangle$. A constraint atom of the form $\langle c \,|\, p(\vec{X})\rangle$ is *projected* when $\mathcal{V}ar(c) \subseteq \{\vec{X}\}$.

We consider the usual operational semantics given in terms of *derivations* from states to states. Let $\langle d \,|\, p(\vec{u}), \vec{B}\rangle$ be a state and $p(\vec{s}) \leftarrow c \wedge \vec{B'}$ be a fresh copy of a rule r. When $solv(\vec{s} = \vec{u} \wedge c \wedge d) = \texttt{true}$ then

$$\langle d \,|\, p(\vec{u}), \vec{B}\rangle \underset{r}{\Longrightarrow} \langle \vec{s} = \vec{u} \wedge c \wedge d \,|\, \vec{B'}, \vec{B}\rangle$$

is a *derivation step* of $\langle d \,|\, p(\vec{u}), \vec{B}\rangle$ w.r.t. r with $p(\vec{s}) \leftarrow c \wedge \overrightarrow{B'}$ as its *input rule*. Let S be the state $\langle d \,|\, \vec{B}\rangle$. S is *failed* if $\vec{B}$ is not empty and no derivation step is possible. S is *successful* if $\vec{B}$ is empty.

We write $S \overset{+}{\underset{P}{\Longrightarrow}} S'$ to summarize a finite number (> 0) of derivation steps from S to S' where each input rule comes from program P. Let S_0 be a state. A sequence of derivation steps $S_0 \underset{r_1}{\Longrightarrow} S_1 \underset{r_2}{\Longrightarrow} \cdots$ of maximal length is called a *derivation* of $P \cup \{S_0\}$ when $r_1, r_2, \ldots$ are rules from P and the *standardization apart* condition holds, *i.e.* each input rule used is variable disjoint from the initial state S_0 and from the input rules used at earlier steps.

3 Concolic Testing

In this section, we briefly introduce concolic testing for both logic programs and CLP.

3.1 Concolic Testing in Logic Programming

We first summarize the framework for concolic testing of logic programs introduced in [16]. On the positive side, in logic programming, the same principle for standard execution, SLD resolution, can also be used for symbolic execution. On the negative side, computing alternative test cases is way more complex than in the traditional setting (e.g., for imperative programs) due to the nondeterministic nature of logic programming computations.

Concolic execution combines both concrete and symbolic execution. However, despite the fact that the concrete and symbolic execution mechanisms are the same, one still needs to consider *concolic* states that combine both a concrete and a symbolic (less instantiated) goal. The concolic execution semantics deals with nondeterminism and backtracking explicitly, similarly to the *linear* operational semantics of [19] for Prolog. In this context, rather than considering a goal, the semantics considers a sequence of goals that, roughly, represents a frontier of the execution tree built so far.

Concolic states have the form $\langle S \,[\![\, S'\rangle$, where S and S' are sequences of (possibly labeled) concrete and symbolic goals, respectively. The structure of S and S' is identical, the only difference being that the atoms in S' might be less instantiated.

Given an arbitrary atom $p(\vec{u})$, an *initial concolic state* has the form $\langle p(\vec{u})_{id} \,[\![\, p(\vec{X})_{id}\rangle$, where $\vec{X}$ are different fresh variables and the labels id denote an initial (empty) computed substitution. Here, $p(\vec{u})$ can be considered a test case (a concrete goal), while $p(\vec{X})$ is the corresponding call with unknown, symbolic arguments, that we use to collect the *constraints* (here: substitutions for $\vec{X}$) using symbolic execution.

Example 3. Given a concrete (atomic) goal, $p(f(X))$, the corresponding initial concolic state has the form $\langle p(f(X))_{id} \,[\![\, p(N)_{id}\rangle$, where N is a fresh variable.

In the following, we assume that every clause c has a corresponding unique label, which we denote by $\ell(c)$. By abuse of notation, we denote by $\ell(\vec{c})$ the set of labels $\{\ell(c_1), \ldots, \ell(c_n)\}$, where $\vec{c} = c_1, \ldots, c_n$. Also, given an atom A and a logic program P, $\mathsf{clauses}(A, P)$ returns the sequence of renamed apart program clauses of P whose head unifies with A.

The semantics is given by the rules of the labeled transition relation $\rightsquigarrow$ shown in Figure 1. Let us briefly explain some aspects of the concolic execution semantics:

$$(success)\ \frac{}{\langle \mathsf{true}_\delta \,|\, S \,[\![\, \mathsf{true}_\theta \,|\, S' \rangle \rightsquigarrow_\diamond \langle \textsc{success}_\delta \,[\![\, \textsc{success}_\theta \rangle}$$

$$(failure)\ \frac{}{\langle (\mathsf{fail}, \vec{B})_\delta \,[\![\, (\mathsf{fail}, \vec{B'})_\theta \rangle \rightsquigarrow_\diamond \langle \textsc{fail}_\delta \,[\![\, \textsc{fail}_\theta \rangle}$$

$$(backtrack)\ \frac{S \neq \epsilon}{\langle (\mathsf{fail}, \vec{B})_\delta \,|\, S \,[\![\, (\mathsf{fail}, \vec{B'})_\theta \,|\, S' \rangle \rightsquigarrow_\diamond \langle S \,[\![\, S' \rangle}$$

$$(choice)\ \frac{\mathsf{clauses}(A,P) = \vec{c} \wedge \vec{c} = \{c_1, \ldots, c_n\} \wedge n > 0 \wedge \mathsf{clauses}(A', P) = \vec{d}}{\langle (A, \vec{B})_\delta \,|\, S \,[\![\, (A', \vec{B'})_\theta \,|\, S' \rangle \rightsquigarrow_{c(\ell(\vec{c}), \ell(\vec{d}))} \langle (A, \vec{B})^{c_1}_\delta \,|\, \ldots \,|\, (A, \vec{B})^{c_n}_\delta \,|\, S \,[\![\, (A', \vec{B'})^{c_1}_\theta \,|\, \ldots \,|\, (A', \vec{B'})^{c_n}_\theta \,|\, S' \rangle}$$

$$(choice\_fail)\ \frac{\mathsf{clauses}(A,P) = \emptyset \wedge \mathsf{clauses}(A', P) = \vec{c}}{\langle (A, \vec{B})_\delta \,|\, S \,[\![\, (A', \vec{B'})_\theta \,|\, S' \rangle \rightsquigarrow_{c(\emptyset, \ell(\vec{c}))} \langle (\mathsf{fail}, \vec{B})_\delta \,|\, S \,[\![\, (\mathsf{fail}, \vec{B'})_\theta \,|\, S' \rangle}$$

$$(unfold)\ \frac{\mathsf{mgu}(A, H_1) = \sigma \wedge \mathsf{mgu}(A', H_1) = \sigma'}{\langle (A, \vec{B})^{H_1 \leftarrow \vec{B_1}}_\delta \,|\, S \,[\![\, (A', \vec{B'})^{H_1 \leftarrow \vec{B_1}}_\theta \,|\, S' \rangle \rightsquigarrow_\diamond \langle (\vec{B_1}\sigma, \vec{B}\sigma)_{\delta\sigma} \,|\, S \,[\![\, (\vec{B_1}\sigma', \vec{B'}\sigma')_{\theta\sigma'} \,|\, S' \rangle}$$

Fig. 1. Concolic execution semantics

- The first rules, *success* and *failure*, use fresh constants labeled with a computed substitution to denote a final state: $\langle \textsc{success}_\delta \rangle$ and $\langle \textsc{fail}_\delta \rangle$, respectively.[4] Note that we are interested in both sucessful and (finitely) failing derivations. Rule *backtrack* applies when the first goal in the sequence finitely fails, but there is at least one alternative choice. In these three rules, we deal with the concrete and symbolic components of the concolic state in much the same way. Also, the steps are labeled with an *empty* label "$\diamond$".
- Rule *choice* represents the first stage of an SLD resolution step. If there is at least one clause whose head unifies with the leftmost atom of the concrete goal, this rule introduces as many copies of a goal as clauses returned by function clauses. Moreover, we label each copy of the goal $(A, \vec{B})$ with a matching clause. If there is at least one matching clause, unfolding is then performed by rule *unfold* using the clause labeling the goal. Otherwise, if there is no matching clause, rule *choice_fail* returns *fail* so that either rule *failure* or *backtrack* applies next.
- Essentially, one can say that the application of rules *choice* and *unfold* amounts to an unfolding step with plain SLD resolution. However, this is only true for the concrete component of the concolic state. Note that, regarding the symbolic component, we do not consider all matching clauses, $\vec{d}$, but only the clauses matching the concrete goal. This is a well known behavior in concolic execution: symbolic execution is restricted to only mimic the steps of the corresponding concrete execution.
- Another relevant point here is that the steps with rules *choice* and *choice_fail* are labeled with a term of the form $c(L_1, L_2)$, where L_1 are the labels of the clauses matching the selected atom in the concrete goal and L_2 are the labels of the clauses matching the selected atom in the corresponding symbolic goal. Note that $L_1 \subseteq L_2$ since the concrete goal is always an instance of the symbolic goal. These labels are essential to compute alternative test cases during concolic testing, as we will see

[4] We note that the semantics only considers the computation of the first solution for the initial goal. This is the way most Prolog applications are used and, thus, the semantics models this behaviour in order to consider a realistic scenario.

$$\begin{array}{lll}
\langle p(f(X))_{id} \,[\![\, p(N)_{id}\rangle & \leadsto^{choice}_{c(\{\ell_1,\ell_2\},\{\ell_1,\ell_2,\ell_3\})} & \langle p(f(X))^{p(f(a))\cdot}_{id} \,|\, p(f(X))^{p(f(b))\cdot}_{id} \;[\![\; p(N)^{p(f(a))\cdot}_{id} \,|\, p(N)^{p(f(b))\cdot}_{id}\rangle \\
& \leadsto^{unfold}_{\diamond} & \langle \mathsf{true}_{\{X/a\}} \,|\, p(f(X))^{p(f(b))\cdot}_{id} \;[\![\; \mathsf{true}_{\{N/f(a)\}} \,|\, p(N)^{p(f(b))\cdot}_{id}\rangle \\
& \leadsto^{success}_{\diamond} & \langle \text{SUCCESS}_{\{X/a\}} \;[\![\; \text{SUCCESS}_{\{N/f(a)\}}\rangle
\end{array}$$

Fig. 2. Concolic execution for $\langle p(f(X))_{id} \,[\![\, p(N)_{id}\rangle$

below. We note that, by collecting these labels, we get an information that shares some similarities with the notion of *characteristic tree* [5].

Example 4. Consider the following logic program:

(ℓ_1) $p(f(a))$.
(ℓ_2) $p(f(b))$.
(ℓ_3) $p(c)$.

and the initial concolic state $\langle p(f(X))_{id} \,[\![\, p(N)_{id}\rangle$. Concolic execution proceeds as shown in Figure 2.

Concolic testing aims at computing "test cases" (concrete atomic goals in our context) that cover all execution paths. Of course, since the number of paths is often infinite, one should consider a timeout or some other method to ensure the termination of the process. Note that concolic testing methods are typically incomplete. A concolic testing algorithm should follow these steps:

1. Given a concrete goal, we construct the associated initial concolic state and run concolic execution. We assume that concrete goals are terminating and, thus, this step is always finite too.
2. Then, we consider each application of rule *choice* in this concolic execution. Consider that the step is labeled with $c(L_1, L_2)$. Here, we are interested in looking for instances of the symbolic goal that match the clauses of every set in $\mathcal{P}(L_2) \setminus L_1$ since the set L_1 is already considered by the current execution.[5]
3. Checking the feasibility for each set in $\mathcal{P}(L_2) \setminus L_1$ is done as follows. Let A be the selected atom in the *symbolic* goal and let $L \in \mathcal{P}(L_2) \setminus L_1$ be the considered set of clauses. Let $\mathcal{H}^+$ be the atoms in the heads of the clauses in L (i.e., the clauses we want to unify with) and let $\mathcal{H}^-$ be the atoms in the heads of the clauses in $L_2 \setminus L$ (i.e., the clauses we do not want to unify with). Then, we are looking for a substitution, θ, such that $A\theta$ unifies with each atom in $\mathcal{H}^+$ but it does not unify with any atom in $\mathcal{H}^-$. This is what we call a *selective unification problem* (see Section 4.1). Usually, we also add another constraint: some variables must become ground by θ. This last requirement is needed to ensure that $A\theta$ is indeed a valid *concrete* (run time) goal where some (input) arguments become ground and, thus, its execution terminates.
4. Finally, for each selective unification problem which is solvable, we have a new concrete goal (i.e., a new test case) and the process starts again. Moreover, one should keep track of the concrete goals already considered and the paths already explored in order to avoid computing the same test case once and again.

Let us now illustrate the concolic testing procedure with a simple example.

Example 5. Consider again the program of Example 4, together with the initial goal $p(f(X))$. For simplicity, we will not consider a groundness condition in this example. Let us start with the concolic execution shown in Figure 2.

[5] For simplicity, we often use the label of a clause to refer to the clause itself.

Given the label $c(\{\ell_1,\ell_2\},\{\ell_1,\ell_2,\ell_3\})$, we have to consider the sets in $\mathcal{P}(\{\ell_1,\ell_2,\ell_3\}) \setminus \{\ell_1,\ell_2\} =$, i.e.,

$$\{\emptyset,\{\ell_1\},\{\ell_2\},\{\ell_3\},\{\ell_1,\ell_3\},\{\ell_2,\ell_3\},\{\ell_1,\ell_2,\ell_3\}\}$$

Thefore, our first selective unification problem, associated to the empty set, considers the atom $p(N)$ and the sets $\mathcal{H}^+ = \emptyset$ and $\mathcal{H}^- = \{p(f(a)), p(f(b)), p(c)\}$. A solution is, e.g., $\{N/a\}$ and, thus, $p(N)\{N/a\} = p(a)$ is another test case to consider.

As for the second set, $\{\ell_1\}$, the selective unification problem considers the atom $p(N)$ and the sets $\mathcal{H}^+ = \{p(f(a))\}$ and $\mathcal{H}^- = \{p(f(b)), p(c)\}$. Here, the only solution is $\{N/f(a)\}$ and, thus, the atom $p(N)\{X/f(a)\} = p(f(a))$ is the next test case to consider.

The process goes on producing the test cases $p(f(b))$ (for the set $\{\ell_2\}$), $p(c)$ (for the set $\{\ell_3\}$), and $p(N)$ (for the set $\{\ell_1,\ell_2,\ell_3\}$), being the remaining problems unfeasible.

3.2 Concolic Testing in CLP

Extending the concolic testing framework from logic programs to CLP is not difficult. In this section, we will focus on the main differences, and will also show an example that illustrates the technique in this setting.

First, the concolic execution semantics for the CLP case is basically equivalent to that in Figure 1 by replacing goals with states of the form $\langle c \,|\, \vec{B}\rangle$ and by considering the usual unfolding rule for CLP programs. Furthermore, the function $\mathsf{clauses}$ is now redefined as follows. Given a state $\langle d \,|\, p(\vec{u})\rangle$ and a set of rules P, we have

$$\begin{aligned}&\mathsf{clauses}(\langle d \,|\, p(\vec{u})\rangle, P)\\ &\quad = \{p(\vec{s}) \leftarrow c \wedge \vec{B} \in P \mid \mathit{solv}(\vec{s} = \vec{u} \wedge c \wedge d) = \mathtt{true}\}\end{aligned}$$

Concolic testing then proceeds basically as in the logic programming case. The main difference, though, is that the selective unification problems now deal with states and CLP programs rather than goals and logic programs. Let us consider a choice step labeled with $c(L_1, L_2)$. Here, given a set $L \in \mathcal{P}(L_2) \setminus L_1$, the sets $\mathcal{H}^+$ and $\mathcal{H}^-$ are built as follows:

$$\begin{aligned}\mathcal{H}^+ &= \{\langle c \,|\, H\rangle \mid H \leftarrow c \wedge \vec{B} \in L\}\\ \mathcal{H}^- &= \{\langle c \,|\, H\rangle \mid H \leftarrow c \wedge \vec{B} \in L_2 \setminus L\}\end{aligned}$$

The groundness condition, if any, will now require some variables to have a fixed value in a given constraint (see Section 4.2).

Example 6. Consider the following CLP($\mathcal{Q}_{lin}$) program:

$$\begin{aligned}&(\ell_1)\ p(X) \leftarrow X \leq 0.\\ &(\ell_2)\ p(X) \leftarrow X \geq 0 \wedge X < 10.\end{aligned}$$

and a choice step labeled with $c(\{\ell_1\},\{\ell_1,\ell_2\})$, where the symbolic state is $\langle \mathsf{true} \,|\, p(N)\rangle$. Hence, we have to consider the sets in $\mathcal{P}(\{\ell_1,\ell_2\}) \setminus \{\ell_1\} =$, i.e.,

$$\{\emptyset,\{\ell_2\},\{\ell_1,\ell_2\}\}$$

Thefore, our first selective unification problem, associated to the empty set, considers the state $\langle \mathsf{true} \,|\, p(N)\rangle$ and the sets

$$\mathcal{H}^+ = \emptyset$$
$$\mathcal{H}^- = \{\langle X \leq 0 \,|\, p(X)\rangle, \langle X \geq 0 \wedge X < 10 \,|\, p(X)\rangle\}$$

A solution is, e.g., $N \geq 10$ and, thus, $\langle N \geq 10 \,|\, p(N)\rangle$ is another test case to consider.

As for the second set, $\{\ell_2\}$, the selective unification problem considers the state $\langle \mathsf{true} \,|\, p(N)\rangle$ and the sets

$$\mathcal{H}^+ = \{\langle X \geq 0 \wedge X < 10 \,|\, p(X)\rangle\}$$
$$\mathcal{H}^- = \{\langle X \leq 0 \,|\, p(X)\rangle\}$$

Here, a possible solution is $N > 0 \wedge N < 10$ and, thus, the state $\langle N > 0 \wedge N < 10 \,|\, p(N)\rangle$ is the next test case to consider.

Finally, for the set $\{\ell_1, \ell_2\}$, the selective unification problem considers the state $\langle \mathsf{true} \,|\, p(N)\rangle$ and the sets

$$\mathcal{H}^+ = \{\langle X \leq 0 \,|\, p(X)\rangle, \langle X \geq 0 \wedge X < 10 \,|\, p(X)\rangle\}$$
$$\mathcal{H}^- = \emptyset$$

where the only solution is $N = 0$ and, thus, our final test case is the state $\langle N = 0 \,|\, p(N)\rangle$.

4 The Selective Unification Problem

In this section, we consider the *selective unification problem*, first in the logic programming (LP) framework [14], as we introduced it in [16, 17], and then in the CLP framework.

4.1 The LP Definition

Let us recall the selective unification problem from the LP setting, together with a few examples. We write $A_1 \approx A_2$ to denote that the atoms A_1 and A_2 unify for some substitution. The *restriction* $\theta\restriction_V$ of a substitution θ to a set of variables V is defined as follows: $x\theta\restriction_V = x\theta$ if $x \in V$ and $x\theta\restriction_V = x$ otherwise.

Definition 1 (selective unification problem). *Let A be an atom, with $G \subseteq \mathcal{V}ar(A)$ a set of variables, and let $\mathcal{H}^+$ and $\mathcal{H}^-$ be finite sets of atoms such that all atoms are pairwise variable disjoint and $A \approx B$ for all $B \in \mathcal{H}^+ \cup \mathcal{H}^-$. Then, the* selective unification problem *for A w.r.t. $\mathcal{H}^+$, $\mathcal{H}^-$ and G is defined as follows:*

$$\mathcal{P}(A, \mathcal{H}^+, \mathcal{H}^-, G) = \left\{ \sigma\restriction_{\mathcal{V}ar(A)} \;\middle|\; \begin{array}{l} \forall H \in \mathcal{H}^+ : A\sigma \approx H \\ \wedge\ \forall H \in \mathcal{H}^- : \neg(A\sigma \approx H) \\ \wedge\ G\sigma \text{ is ground} \end{array} \right\}$$

Example 7. We illustrate the notion of selective unification with several examples:

- Let $A = p(X)$, $\mathcal{H}^+ = \{p(a), p(b)\}$, $\mathcal{H}^- = \emptyset$ and $G = \emptyset$. The empty substitution is a solution, as $p(X)$ unifies with $p(a)$ and $p(b)$.
- Let $A = p(X)$, $\mathcal{H}^+ = \{p(a), p(b)\}$, $\mathcal{H}^- = \{p(f(Z))\}$ and $G = \emptyset$. This problem has no solution. One cannot find an instance of A that unifies with both atoms in $\mathcal{H}^+$ and does not unify with $p(f(Z))$.

– Let $A = p(X)$, $\mathcal{H}^+ = \{p(s(Y))\}$, $\mathcal{H}^- = \{p(s(0))\}$ and $G = \{X\}$. There are infinitely many solutions, among them we find $\{X/s^{n+2}(0)\}$ for $n \in \mathbb{N}$. For instance, let us check that $\sigma = \{X/s(s(0))\}$ is a solution. We have $A\sigma = p(s(s(0)))$. $A\sigma$ and $p(s(Y))$ unify while $A\sigma$ and $p(s(0))$ do not unify, and $X\sigma$ is ground.
– Let $A = p(X, Y)$, $\mathcal{H}^+ = \{p(a, b), p(Z, Z)\}$, and $\mathcal{H}^- = \emptyset = G$. There are two solutions, $\{X/a\}$ and $\{Y/b\}$. Let us check that $\sigma = \{X/a\}$ is a solution. We have $A\sigma = p(a, Y)$. $A\sigma$ and $p(a, b)$ unify, and $A\sigma$ and $p(Z, Z)$ unify. $\mathcal{H}^-$ and G are empty so the last two conditions of $\mathcal{P}(A, \mathcal{H}^+, \mathcal{H}^-, G)$ are trivially true.

4.2 The CLP Definition

In this section, we generalize Definition 1 to CLP. Let $A_1 = \langle c_1 \,|\, p(\vec{u})\rangle$ and $A_2 = \langle c_2 \,|\, p(\vec{s})\rangle$ be two constraint atoms with no common variable. We write $A_1 \approx A_2$ or A_1 *unifies with* A_2 to denote that $\mathcal{D} \models \exists(\vec{u} = \vec{s} \wedge c_1 \wedge c_2)$.

For simplicity, in the following definition, we consider that the constraint atom has the form $\langle c_A \,|\, p(\vec{X})\rangle$. There is no loss of generality since any arbitrary constraint atom $\langle c \,|\, p(\vec{u})\rangle$ can be trivially transformed into $\langle \vec{u} = \vec{X} \wedge c \,|\, p(\vec{X})\rangle$.

Definition 2 (constraint selective unification problem (CSUP)). *Let A be a constraint atom of the form $\langle c_A \,|\, p(\vec{X})\rangle$ with $G \subseteq \mathcal{V}ar(A)$. Let $\mathcal{H}^+$ and $\mathcal{H}^-$ be finite sets of constraint atoms such that all constraint atoms, including A, are pairwise variable disjoint and $A \approx B$ for all $B \in \mathcal{H}^+ \cup \mathcal{H}^-$. Then, the* constraint selective unification problem *for A w.r.t. $\mathcal{H}^+$, $\mathcal{H}^-$ and G is defined as follows:*
$\mathcal{P}(A, \mathcal{H}^+, \mathcal{H}^-, G) =$

$$\left\{ c_A \wedge c \;\middle|\; \begin{array}{l} \quad c_A \wedge c \text{ is satisfiable} \\ \wedge\ c \text{ is variable disjoint with } \mathcal{H}^+ \cup \mathcal{H}^- \\ \wedge\ \forall H \in \mathcal{H}^+ : \langle c_A \wedge c \,|\, p(\vec{X})\rangle \approx H \\ \wedge\ \forall H \in \mathcal{H}^- : \neg(\langle c_A \wedge c \,|\, p(\vec{X})\rangle \approx H) \\ \wedge\ \text{each } X \in G \text{ is fixed within } c_A \wedge c \end{array} \right\}$$

Intuitively, to solve a CSUP, we consider any constraint c such that $\langle c_A \wedge c \,|\, p(\vec{X})\rangle$ still unifies with all the positive constraint atoms while preventing any unification with the negative constraint atoms and ensuring that the variables in G have a fixed value.

Note that $X \in G$ *is fixed within* $c_A \wedge c$ is the equivalent of the *groundness* condition of the LP case and we keep calling it the *groundness* condition in the CLP case. Some constraint solvers might give to X the required value, but it is not mandatory as it can be expressed in first order logic by stating that, within $c_A \wedge c$, X has exactly one value. E.g., X is fixed within $c(X, \vec{X})$ is equivalent to $\mathcal{D} \models \exists X[\exists \vec{X} c(X, \vec{X}) \wedge [\forall Y \forall \vec{Y}(c(Y, \vec{Y}) \rightarrow X = Y]]$.

Example 8 (CLP($\mathcal{Q}_{lin}$)). We illustrate the notion of selective unification in the context of CLP($\mathcal{Q}_{lin}$) with several examples:

– Let $A := \langle 0 \leq X \wedge X \leq 5 \,|\, p(X)\rangle$, $\mathcal{H}^+ := \{\langle 4 \leq Y \,|\, p(Y)\rangle\}$, $\mathcal{H}^- := \{\langle Z < 2 \,|\, p(Z)\rangle\}$, and $G = \{X\}$. There is an infinite number of solutions to $\mathcal{P}(A, \mathcal{H}^+, \mathcal{H}^-, G)$, among which one finds the constraint $X = 9/2$. It is equivalent to the satisfiable constraint $0 \leq X \wedge X \leq 5 \wedge X = 9/2$, the constraint $X = Y, 4 \leq Y, X = 9/2$ is satisfiable while $X = Z, Z < 2, X = 9/2$ is unsatisfiable, and it entails that X is ground.

– Let $A := \langle 0 \leq X \wedge X \leq 5 \,|\, p(X)\rangle$, $\mathcal{H}^+ := \{\langle 4 \leq Y_1 \,|\, p(Y_1)\rangle, \langle Y_2 \leq 1 \,|\, p(Y_2)\rangle\}$, $\mathcal{H}^- := \{\langle 2 < Z \wedge Z < 3 \,|\, p(Z)\rangle\}$, and $G = \emptyset$. There is no solution to $\mathcal{P}(A, \mathcal{H}^+, \mathcal{H}^-, G)$. Intuitively, if there is a solution on the left of 2, then it excludes the first positive atom or if there is a solution to the right of 3, then it excludes the second positive atom. So one cannot find a conjunction of atomic constraints that include the elements of $\mathcal{H}^+$ and exclude the element of $\mathcal{H}^-$ because the set of points described by such a conjunction is convex.

5 Undecidability of the CSUP

In this section, we show that solving the constraint selective unification problem in CLP is generally undecidable. We consider any Turing machine M and any word w. We define an instance $\mathcal{P}_{M,w}$ of the generic CSUP in a constraint logic programming language CLP($\mathcal{A}$), the class of constraints of which is a strict subclass of the *array property* fragment introduced in [1]. We encode the tape of M as an array and we define $\mathcal{P}_{M,w}$ so that M accepts w if and only if $\mathcal{P}_{M,w}$ has a solution. Section 5.1 gives the exact definition of CLP($\mathcal{A}$). Section 5.2 introduces notations and definitions about Turing machines. Section 5.3 describes the reduction of the CSUP to the acceptance problem for Turing machines.

5.1 CLP($\mathcal{A}$)

In order to show the undecidability of the CSUP, we consider a strict subclass of the array property fragment: we do not use array writes and we only handle arrays, the indices and elements of which are naturals (indices in [1] are integers). Therefore, our constraint logic programming language CLP($\mathcal{A}$) is defined as follows.

– $\Sigma_\mathcal{A} := \Sigma_\mathbb{N} \cup \{+, =, \leq\} \cup \{\cdot[\cdot]\}$ is the constraint domain signature where $\Sigma_\mathbb{N}$ consists of all the sequences of digits and $\cdot[\cdot]$ is the *array read*. The read $a[i]$ returns the value stored at position i of the array a. For multidimensional arrays, we abbreviate $a[i]\cdots[j]$ with $a[i, \ldots, j]$.
– The class of constraints $\mathcal{L}_\mathcal{A}$ consists of all existentially-closed Boolean combinations of *index atoms*, *element atoms* and *array properties*, defined as follows. Index and element terms have sort $\mathbb{N}$. Index terms are constructed from $\Sigma_\mathbb{N} \cup \{+\}$ and index variables. Array terms have functional sorts:
  - One-dimensional sort: $\mathbb{N} \to \mathbb{N}$
  - Multidimensional sort: $\mathbb{N} \to \cdots \to \mathbb{N}$;
    e.g., a two-dimensional array has sort $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$.

  For array term a and index term i, $a[i]$ is either an element term if a has sort $\mathbb{N} \to \mathbb{N}$, or an array term if a has a multidimensional sort. The only element terms we consider are the constant symbols in $\Sigma_\mathbb{N}$ and those terms that have the form $a[i]$ where a is an array term of sort $\mathbb{N} \to \mathbb{N}$ and i is an index term. An *index atom* has the form $i = i'$ or $i \leq i'$ where i and i' are index terms. An *element atom* has the form $t = t'$ or $t \leq t'$ where t and t' are element terms. Our *array properties* are simpler than in [1]: they have the form

  $$(\forall I)(\varphi_i(I) \to \varphi_v(I))$$

  where I is an index variable and $\varphi_i(I)$ and $\varphi_v(I)$ are the *index guard* and the *value constraint*, respectively. The index guard follows the grammar:

  $$\begin{aligned} &\mathit{iguard} : \mathit{iterm} \leq \mathit{iterm} \\ &\mathit{iterm} : I \mid \text{an index term in which } I \text{ does not occur} \end{aligned}$$

The value constraint $\varphi_v(I)$ is a disjunction of element atoms where I can only be used in array read expressions of the form $a[I]$. Array reads cannot be nested; e.g., $a[a'[I]]$ is not allowed.

Example 9. Let T, T' be some array variables and I be an index variable. Then, $T[0] = 1$ and $T[0] = T'[0]$ are element atoms. The formula $(\forall I)(2 \leq I \rightarrow T[I] = T'[I])$ is an array property expressing that arrays T and T' are equal from index 2. The formula $(\exists T, T')\varphi$ where

$$\varphi := \big(T[0] = 1 \wedge \neg(T[0] = T'[0]) \\ \wedge(\forall I)(2 \leq I \rightarrow T[I] = T'[I])\big)$$

is a constraint in $\mathcal{L}_\mathcal{A}$.

We use $\cdot \neq \cdot$ as a shorthand for $\neg(\cdot = \cdot)$. Observe that equality between arrays is not permitted. Equality between a and a' of sort $\mathbb{N} \rightarrow \mathbb{N}$ can be written as the array property $(\forall I)(a[I] = a'[I])$, the index guard of which is the always satisfiable formula true.

- $\mathcal{D}_\mathcal{A}$ is the $\Sigma_\mathcal{A}$-structure whose domain $D_\mathcal{A}$ consists of the naturals and of the arrays of naturals of any dimension, and where the symbols in $\Sigma_\mathbb{N} \cup \{+, =, \leq\}$ are interpreted as usual over the naturals and the function symbol $\cdot[\cdot]$ is interpreted as the array read as indicated above. We use the $[\cdot]$ notation for denoting arrays in $D_\mathcal{A}$; e.g., $[4, 0, 3]$ denotes the array which consists of the natural 4 at index 0, the natural 0 at index 1 and the natural 3 at index 2.

  Example 10 (Example 9 continued). Let $a := [1, 0, 5, 5]$, $a' := [6, 9, 5, 5]$ and $a'' := [6, 9, 5, 4]$. Let v be a valuation with $v(T) = a$ and $v(T') = a'$. Then, we have $\mathcal{A} \models_v \varphi$. But $\mathcal{A} \models_{v'} \neg\varphi$ for any valuation v' with $v'(T) = a$ and $v'(T') = a''$.

- A constraint solver $solv_\mathcal{A}$ for $\mathcal{L}_\mathcal{A}$ always returning either `true` or `false` can be built from the decision procedure presented in [1].

The signature used for writing constraint atoms is $\Sigma_\mathcal{A} \cup \{p\}$ where p is a single-argument predicate symbol.

5.2 Turing Machines

We adhere to the notations and definitions presented in [8].

Let $M := (Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$ be a (deterministic) Turing machine:

- Q is the finite set of *states*,
- Γ is the finite set of allowable *tape symbols*, with $Q \cap \Gamma = \emptyset$,
- $\Box$, a symbol of Γ, is the *blank*,
- Σ, a subset of Γ not including $\Box$, is the set of *input symbols*,
- $\delta : (Q \setminus F) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is a partial function called the *next move function* (L stands for *left* and R for *right*),
- q_0 in $Q \setminus F$ is the *start state*,
- $F \subseteq Q$ is the set of *final states*.

Elements of Q are denoted by $q, q_0, \ldots$, elements of Σ by $e, e_0, \ldots$ and elements of Γ by $x, y, x_0, y_0, \ldots$ We assume that M consists of a single input tape that is divided into cells and that is infinite to the left and to the right. An *instantaneous description* (ID) of M has the form $\alpha_1 q \alpha_2$. Here q, the current state of M, is in Q; $\alpha_1\alpha_2$ is the string in Γ^* that is the current content of the tape from the leftmost nonblank

symbol up to the rightmost nonblank symbol (observe that the blank may occur in $\alpha_1\alpha_2$). The tape head is assumed to be scanning the leftmost symbol of α_2, or if $\alpha_2 = \epsilon$, the head is scanning a blank. A *move* of M has the form $id_1 \vdash id_2$ where id_1 and id_2 are ID's of M and $\vdash$ is the transition relation of M derived from δ. A sequence $id_0, \ldots, id_m$ of ID's is a *computation* of M if $id_i \vdash id_{i+1}$ holds for every i in $[0, m-1]$. It is an *invalid computation* of M if an invalid move occurs in it, i.e., $m \geq 1$ and $id_i \nvdash id_{i+1}$ holds for some i in $[0, m-1]$. We say that M *accepts* an input $w \in \Sigma^*$ when there exists a finite computation $id_0, \ldots, id_m$ of M where $id_0 = q_0 w$ and the state in id_m is final. Note that δ is undefined for any (q, x) with $q \in F$. Hence M halts, i.e., has no next move, whenever the input is accepted. Moreover, the start state is not final.

5.3 Reduction to the Acceptance Problem

Let $w := e_0 \cdots e_l$ be a word in Σ^*. From M and w we construct an instance $\mathcal{P}_{M,w}$ of the generic CSUP $\mathcal{P}$ in the language CLP($\mathcal{A}$). Without loss of generality, we suppose that $Q \cup \Gamma \subseteq \mathbb{N}$. We represent an ID $x_0 \ldots x_n q y_0 \ldots y_m$ by arrays a satisfying the following constraints: $a[0]$ is an array that starts with q; $a[1]$ is an array that represents the portion of the tape to the left of the head: it starts with $x_n, \ldots, x_0$ (the word $x_0 \ldots x_n$ in reverse order), followed by blanks (possibly none); $a[2]$ is an array that represents the portion of the tape to the right of the head: it starts with $y_0, \ldots, y_m$, followed by blanks (possibly none). We represent a sequence $id_0, \ldots, id_m$ of ID's by any array that starts with $a_0, \ldots, a_m$, where for each i in $[0, m]$, a_i is an array representing id_i.

Example 11. Let us consider $Q := \{q_0, q_1, q_2, q_3, q_4\}$, $\Sigma := \{0, 1\}$, $\Gamma := \{0, 1, x, y, \Box\}$, $F := \{q_4\}$ and the next move function δ:

δ	0	1	x	y	$\Box$
q_0	(q_1, x, R)			(q_3, y, R)	
q_1	$(q_1, 0, R)$	(q_2, y, L)		(q_1, y, R)	
q_2	$(q_2, 0, L)$		(q_0, x, R)	(q_2, y, L)	
q_3				(q_3, y, R)	$(q_4, \Box, R)$

Then M accepts the language $L = \{0^n 1^n \mid n \geq 1\}$. The array

$$\begin{array}{l}\big[[[q_0], [], [0, 1, \Box]],\ [[q_1], [x], [1, \Box]],\ [[q_2], [], [x, y, \Box]],\\ \ [[q_0], [x], [y, \Box]],\ [[q_3], [y, x], [\Box]],\ [[q_4], [\Box, y, x], []]\big]\end{array}$$

represents the computation

$$q_0 01 \vdash x q_1 1 \vdash q_2 x y \vdash x q_0 y \vdash x y q_3 \vdash x y \Box q_4$$

while

$$\big[[[q_0], [], [0, 0]],\ [[q_1], [x], [0]],\ [[q_1], [0, x], []]\big]$$

represents the computation $q_0 00 \vdash x q_1 0 \vdash x 0 q_1$.

Definition 3. *Let a be an array term and i, j be some index terms. We define the following constraints.*

- *Empty$(a, i) := (\forall I)(i \leq I \rightarrow a[I] = \Box)$, where I is a fresh index variable, expresses that the tape represented by a is empty from index i.*

– *Start*(a) *expresses that the computation represented by* a *starts with the ID* q_0w*, i.e., at index 0 of* a *there is an array representing* q_0w*:*

$$Start(a) := \begin{pmatrix} a[0,0,0] = q_0 \\ \wedge\ Empty(a[0,1],0) \\ \wedge\ (\wedge_{k\in[0,l]} a[0,2,k] = e_k) \\ \wedge\ Empty(a[0,2],l+1) \end{pmatrix}$$

– $Fin(a,i) := (\vee_{q\in F} a[i,0,0] = q)$ *denotes that there is a final state at index* i *in the computation represented by* a*.*

– $NoFin(a,i) := (\forall I)\left(I \leq i \rightarrow (\vee_{q\in Q\setminus F} a[I,0,0] = q)\right)$*, with* I *a fresh index variable, denotes that in the computation represented by* a *there is no final state before index* i*.*

– $Inv(a,i,j)$ *denotes that in the computation represented by* a *there is an invalid move at index* i*:*

$$Inv(a,i,j) := \begin{pmatrix} (c_R(a,i) \vee d_R(a,i,j)) \\ \wedge\ (c_L(a,i) \vee d_L(a,i,j)) \end{pmatrix}$$

A move to the right $\alpha_1 q x \alpha_2 \vdash \alpha_1' y q' \alpha_2'$ *is valid when* $\alpha_1 = \alpha_1'$ *and* $\alpha_2 = \alpha_2'$ *and* $\delta(q,x) = (q',y,R)$*. So, it is invalid when*

$$\alpha_1 \neq \alpha_1' \text{ or } \alpha_2 \neq \alpha_2' \tag{1}$$

$$\text{or } \forall((q_1,x_1),(q_1',y_1,R)) \in \delta : \begin{cases} q_1 \neq q \\ \text{or } x_1 \neq x \\ \text{or } q_1' \neq q' \\ \text{or } y_1 \neq y \end{cases} \tag{2}$$

The formula $c_R(a,i)$ *expresses condition* (2)*:*

$$c_R(a,i) := \bigwedge_{\delta(q,x)=(q',y,R)} \begin{pmatrix} a[i,0,0] \neq q \\ \vee\ a[i,2,0] \neq x \\ \vee\ a[i+1,0,0] \neq q' \\ \vee\ a[i+1,1,0] \neq y \end{pmatrix}$$

Note that α_1 *is represented by* $a[i,1]$ *(in reverse order) and that* $\alpha_1' y$ *is represented by* $a[i+1,1]$ *(in reverse order). Moreover,* $x\alpha_2$ *is represented by* $a[i,2]$ *and* α_2' *is represented by* $a[i+1,2]$*. So, the formula*

$$d_R(a,i,j) := \begin{pmatrix} a[i,1,j] \neq a[i+1,1,j+1] \\ \vee\ a[i,2,j+1] \neq a[i+1,2,j] \end{pmatrix}$$

denotes that, for some index j*, the* j*-th symbol of* α_1 *is different from that of* α_1' *or the* j*-th symbol of* α_2 *is different from that of* α_2'*. Therefore, it expresses that* $\alpha_1 \neq \alpha_1'$ *or* $\alpha_2 \neq \alpha_2'$*, i.e., it expresses condition* (1)*. Formulas* $c_L(a,i)$ *and* $d_L(a,i,j)$ *are constructed similarly for the moves to the left:*

$$c_L(a,i) := \bigwedge_{\delta(q,x)=(q',y,L)} \begin{pmatrix} a[i,0,0] \neq q \\ \vee\ a[i,2,0] \neq x \\ \vee\ a[i+1,0,0] \neq q' \\ \vee\ a[i+1,2,1] \neq y \end{pmatrix}$$

$$d_L(a,i,j) := \begin{pmatrix} a[i,1,0] \neq a[i+1,2,0] \\ \vee\ a[i,1,j+1] \neq a[i+1,1,j] \\ \vee\ a[i,2,j+1] \neq a[i+1,2,j+2] \end{pmatrix}$$

Definition 4. *Let T, T^+, T^- be array variables and K, I_1, I_2 be index variables. We define $\mathcal{P}_{M,w}$ as follows:*

$$\begin{aligned} A &:= \langle c_A \mid p(T)\rangle \text{ with } c_A := \mathit{Start}(T) \\ G &:= \varnothing \\ \mathcal{H}^+ &:= \{H^+\} \\ \mathcal{H}^- &:= \{H^-\} \end{aligned}$$

with

$$\begin{aligned} H^+ &:= \langle (\exists K)\mathit{Fin}(T^+, K) \mid p(T^+)\rangle \\ H^- &:= \langle (\exists I_1, I_2)(\mathit{Inv}(T^-, I_1, I_2) \wedge \mathit{NoFin}(T^-, I_1)) \mid p(T^-)\rangle \end{aligned}$$

The atom H^+ (resp. H^-) represents all the sequences of ID's that contain a final state (resp. all the invalid computations of M).

Lemma 1. *$\mathcal{P}_{M,w}$ is a valid instance of the generic CSUP $\mathcal{P}$.*

Proof. A is a constraint atom of the form $\langle c_A \mid p(T)\rangle$ with $G \subseteq \mathcal{V}ar(A)$. Moreover, $\mathcal{H}^+$ and $\mathcal{H}^-$ are finite sets of constraint atoms such that all the constraint atoms in $\{A\} \cup \mathcal{H}^+ \cup \mathcal{H}^-$ are pairwise variable disjoint. Let

$$a_0 := \big[[q_0], [], [e_0, \ldots, e_l, \Box]\big]$$

and

$$a_1 := \big[[q_f], [x], [e_1, \ldots, e_l, \Box]\big]$$

with q_f a final state of M and $\delta(q_0, e_0) \neq (q_f, x, R)$ and $\delta(q_0, e_0) \neq (q_f, x, L)$ (if $w = \varepsilon$ then replace e_0 with $\Box$). Let v be a valuation with $v(T) = v(T^+) = v(T^-) = [a_0, a_1]$. Then, the formula $c^+ := \big(T = T^+ \wedge c_A \wedge (\exists K)\mathit{Fin}(T^+, K)\big)$ is such that $\mathcal{A} \models_v c^+$, hence we have $\mathcal{A} \models \exists c^+$, i.e., $A \approx H^+$. Let

$$c^- := \big(T = T^- \wedge c_A \wedge (\exists I_1, I_2) \underbrace{(\mathit{Inv}(T^-, I_1, I_2) \wedge \mathit{NoFin}(T^-, I_1))}_{\varphi}\big) .$$

Let v' be a valuation with $v'(I_1) = v'(I_2) = 0$ and $v'(X) = v(X)$ for any variable X different from I_1 and I_2. Then we have $\mathcal{A} \models_{v'} \mathit{Inv}(T^-, I_1, I_2)$ because $[a_0, a_1]$ represents an invalid move of M. Moreover, $\mathcal{A} \models_{v'} \mathit{NoFin}(T^-, I_1)$ because the state in a_0 is q_0, i.e., is not final. Consequently, we have $\mathcal{A} \models_{v'} \varphi$ so $\mathcal{A} \models_v (\exists I_1, I_2)\varphi$. As we also have $\mathcal{A} \models_v (T = T^- \wedge c_A)$, then $\mathcal{A} \models_v c^-$, i.e., $\mathcal{A} \models \exists c^-$. So, we have $A \approx H^-$.

Proposition 1. *If M accepts w then $\mathcal{P}_{M,w}$ has a solution.*

Proof. Suppose that M accepts w. Then, there exists a finite computation of M of the form $id_0, \ldots, id_m$ where $id_0 = q_0 w$ and the state in id_m is final. For each i in $[0, m]$, suppose that id_i has the form $x_0^i \ldots x_{n_i}^i q^i y_0^i \ldots y_{m_i}^i$ and let

$$c_i := \begin{pmatrix} T[i,0,0] = q^i \\ \wedge\ (\wedge_{j\in[0,n_i]} T[i,1,j] = x^i_{n_i - j}) \wedge \mathit{Empty}(T[i,1], n_i + 1) \\ \wedge\ (\wedge_{j\in[0,m_i]} T[i,2,j] = y^i_j) \wedge \mathit{Empty}(T[i,2], m_i + 1) \end{pmatrix}$$

Note that we have $c_0 = c_A$. Let $c := (c_1 \wedge \cdots \wedge c_m)$.

– $c_A \wedge c$ is satisfiable because it is a conjunction of atomic constraints, each of them "assigns" a value to a distinct cell of T. Moreover, by definition of the c_i's, for any valuation v such that $\mathcal{A} \models_v c_A \wedge c$, we have that $v(T)$ is an array that starts with $a_0, \ldots, a_m$ where for each each i in $[0, m]$, a_i is an array that represents id_i.
– c is variable disjoint with H^+ and H^-.
– As q^m is a final state of M, we have

$$\mathcal{A} \models \exists[T = T^+ \wedge (c_A \wedge c) \wedge (\exists K) Fin(T^+, K)]$$

hence $\langle c_A \wedge c \mid p(T)\rangle \approx H^+$.
– Suppose that $\mathcal{A} \models \exists c^-$ where

$$c^- := \left[\begin{array}{l} T = T^- \wedge (c_A \wedge c) \\ \wedge\ (\exists I_1, I_2)(Inv(T^-, I_1, I_2) \wedge NoFin(T^-, I_1)) \end{array}\right] \tag{3}$$

Then, there is a valuation v which is such that $\mathcal{A} \models_v c^-$. By (3), we have $\mathcal{A} \models_v (T = T^- \wedge (c_A \wedge c))$, hence $v(T^-)$ is an array that starts with $a_0, \ldots, a_m$ where for each each i in $[0, m]$, a_i is an array that represents id_i. By (3), we have $\mathcal{A} \models_v (\exists I_1, I_2)(Inv(T^-, I_1, I_2) \wedge NoFin(T^-, I_1))$, so there exists a valuation v' which is such that $v'(X) = v(X)$ for any variable X different from I_1 and I_2, and

$$\mathcal{A} \models_{v'} (Inv(T^-, I_1, I_2) \wedge NoFin(T^-, I_1)) \,. \tag{4}$$

So, $\mathcal{A} \models_{v'} NoFin(T^-, I_1)$ and as q^m is a final state of M, we necessarily have $v'(I_1) < m$. As $id_0, \ldots, id_m$ is a computation of M, we have $id_{v'(I_1)} \vdash id_{v'(I_1)+1}$, so $\mathcal{A} \models_{v'} \neg Inv(T^-, I_1, I_2)$ and we have a contradiction with (4). Therefore, we have $\mathcal{A} \models \neg\exists c^-$, i.e., $\langle c_A \wedge c \mid p(T)\rangle \not\approx H^-$.
– As G is the empty set, each $X \in G$ is fixed within $c_A \wedge c$.

Consequently, $c_A \wedge c \in \mathcal{P}_{M,w}$.

Proposition 2. *If $\mathcal{P}_{M,w}$ has a solution then M accepts w.*

Proof. Suppose that $\mathcal{P}_{M,w}$ has a solution $c_A \wedge c$. Then, we have $\langle c_A \wedge c \mid p(T)\rangle \approx H^+$, i.e., $\mathcal{A} \models \exists c^+$ where

$$c^+ := \big[T = T^+ \wedge (c_A \wedge c) \wedge (\exists K) Fin(T^+, K)\big] \,. \tag{5}$$

So, there exists a valuation v which is such that $\mathcal{A} \models_v c^+$. Note that we have $T^- \notin \mathcal{V}ar(c^+)$ because $\mathcal{V}ar(c_A) = \{T\}$ and c is variable disjoint with H^- (because $c_A \wedge c \in \mathcal{P}_{M,w}$). Let v' be a valuation with $v'(X) = v(X)$ for all variable X of c^+ and $v'(T^-) = v'(T)$. Then, $\mathcal{A} \models_{v'} c^+$ and, by (5), $\mathcal{A} \models_{v'} c_A \wedge c$. Let $a := v'(T)$.

– We have $\mathcal{A} \models_{v'} c_A$ so, by definition of c_A, a is an array and $a[0]$ is an array that represents the ID $q_0 w$.
– By (5), $v'(T^+) = v'(T) = a$ and $\mathcal{A} \models_{v'} (\exists K) Fin(T^+, K)$. So, there exists an index k which is such that $a[k, 0, 0] \in F$. Let m be the least such index, i.e.,

$$a[m, 0, 0] \in F \quad \text{and} \quad \forall i < m,\ a[i, 0, 0] \notin F \,. \tag{6}$$

– As $c_A \wedge c \in \mathcal{P}_{M,w}$, we have $\langle c_A \wedge c \mid p(T)\rangle \not\approx H^-$. Hence, $\mathcal{A} \models \neg\exists c^-$, i.e., $\mathcal{A} \models \forall\neg c^-$, where

$$c^- := [\ T = T^- \wedge (c_A \wedge c) \\ \wedge(\exists I_1, I_2)(\mathit{Inv}(T^-, I_1, I_2) \wedge \mathit{NoFin}(T^-, I_1))]\ .$$

Consequently, we have $\mathcal{A} \models_{v'} \neg c^-$. Note that

$$\neg c^- = [T \neq T^- \vee \neg(c_A \wedge c) \\ \vee \underbrace{(\forall I_1, I_2)(\neg \mathit{Inv}(T^-, I_1, I_2) \vee \neg \mathit{NoFin}(T^-, I_1))}_{\varphi}]\ .$$

As $\mathcal{A} \models_{v'} (T = T^- \wedge (c_A \wedge c))$, we have necessarily $\mathcal{A} \models_{v'} \varphi$. Let i be a natural which is less than m where m is defined in (6). Let v'' be a valuation which is such that $v''(X) = v'(X)$ for all variable X different from I_1 and $v''(I_1) = i$. Then, we have $v''(T^-) = v'(T^-) = v'(T) = a$ so, by (6), $\mathcal{A} \models_{v''} \mathit{NoFin}(T^-, I_1)$. We have $\mathcal{A} \models_{v''} \neg \mathit{Inv}(T^-, I_1, I_2)$ so $a[i]$ represents an ID id_i, $a[i{+}1]$ represents an ID id_{i+1} and $id_i \vdash id_{i+1}$ is a valid move of M.

Consequently, the array a represents a computation of M starting from $q_0 w$ and ending in a final state, so M accepts w.

It is undecidable whether an arbitrary Turing machine accepts an arbitrary word. Therefore, Proposition 1 and Proposition 2 yield the following result.

Theorem 1. *It is undecidable whether an arbitrary instance of the CSUP has a solution.*

6 A Decidable Case for the CSUP

As Section 5 has shown the undecidability of the generic CSUP, let us introduce additional hypotheses about the constraint structure $\mathcal{D}$ which will help solving the problem:

A1: The constraint structure admits variable elimination.
A2: The negation of any atomic constraint is equivalent to a finite disjunction of atomic constraints.

Example 12. $\mathcal{Q}_{lin}$ verifies these assumptions. It admits variable elimination. The set of predefined atomic constraints is $\{< /2, \leq /2, = /2, \geq /2, > /2\}$. The negation of each atomic constraint is an atomic constraint, except for $= /2$ whose negation is defined by a disjunction of atomic constraints, i.e., $\neg(X = Y) \equiv X < Y \vee X > Y$.

The next subsection will show that assumptions A1 and A2 are sufficient conditions on the constraint domain to solve the CSUP when $G = \emptyset$. Then, by relying on specific properties of $\mathcal{Q}_{lin}$, we present an algorithm which has been implemented solving the CSUP when $G \neq \emptyset$.

6.1 CSUP without the Groundness Condition

In Figure 3, we present an algorithm which we call CSUP$^-$ for solving the constraint selective unification without the groundness condition.

Note that assumption A1 is used in the preconditions of the algorithm as any constraint atom is assumed to be projected. The computation of C' in step 1 is simply a renaming. Step 2 relies explicitly on assumption A2 and implicitly on assumption A1, because each disjunct is a constraint involving only $\vec{X}$.

Preconditions: Let A be a projected constraint atom $\langle c_A \,|\, p(\vec{X})\rangle$. Let $\mathcal{H}^+$ and $\mathcal{H}^-$ be finite sets of constraint atoms such that all constraint atoms, including A, are pairwise variable disjoint and $A \approx B$ for all $B \in \mathcal{H}^+ \cup \mathcal{H}^-$.

Postcondition: A finite set of constraints, each of them being a solution of $\mathcal{P}(A, \mathcal{H}^+, \mathcal{H}^-, \emptyset)$.

Algorithm:

(1) Intersect of all the complements of the atoms in $\mathcal{H}^-$:

$$I := \bigwedge\{\neg C' | H = \langle c' | p(\vec{Y})\rangle \in \mathcal{H}^-, C' \equiv \exists \vec{Y}[\vec{X} = \vec{Y} \wedge c']\}$$

(2) Eliminate negation from I then distribute $\wedge$ over $\vee$:

$$J := \bigvee_{1 \leq j \leq n} C_j(\vec{X})$$

(3) Intersect J with A. Let $C_j^A(\vec{X}) \equiv [C_j(\vec{X}) \wedge c_A]$for $1 \leq j \leq n$ in:

$$K := \bigvee_{1 \leq j \leq n} [C_j^A(\vec{X})]$$

(4) Collect the constraints from K which intersect each of $\mathcal{H}^+$:

$$S := \left\{ C_j^A(\vec{X}) \in K | \bigwedge_{\langle c' | p(\overrightarrow{X'})\rangle \in \mathcal{H}^+} \mathcal{D} \models \exists[\overrightarrow{X'} = \vec{X} \wedge C_j^A(\vec{X}) \wedge c'] \right\}$$

(5) Return S

Fig. 3. Algorithm CSUP$^-(A, \mathcal{H}^+, \mathcal{H}^-)$.

Theorem 2. *Algorithm CSUP$^-$ terminates, is correct and complete.*

Proof.

- Each step 1–5 terminates, so termination is obvious.
- Correctness. Let $C_j^A(\vec{X})$ be an element of S. We show that $C_j^A(\vec{X}) \in \mathcal{P}(A, \mathcal{H}^+, \mathcal{H}^-, \emptyset)$. Let H denote the constraint atom $\langle C_j^A(\vec{X}) \,|\, p(\vec{X})\rangle$.
  - $C_j^A(\vec{X})$ is a finite satisfiable conjunction of atomic constraints because of steps 2–4 of CSUP$^-$.
  - As defined in step 3, $C_j^A(\vec{X}) \equiv [C_j(\vec{X}) \wedge c_A]$.
  - $\mathcal{V}ar(C_j(\vec{X})) \subseteq \vec{X}$, so the added constraint is variable disjoint with $\mathcal{H}^+ \cup \mathcal{H}^-$.
  - H unifies with all constraint atom from $\mathcal{H}^+$ because of step 4.
  - H does not unify with any constraint atom from $\mathcal{H}^-$ because of step 1.
- Completeness. Let c be any element of $\mathcal{P}(A, \mathcal{H}^+, \mathcal{H}^-, \emptyset)$ and $c(\vec{X})$ its projection onto $\vec{X}$. As each of the first four steps of the algorithm is required by the problem statement and is dealt with by logical equivalence, c is covered by the union of C_j^A, i.e., $\mathcal{D} \models \forall \vec{X}[c(\vec{X}) \rightarrow \cup_{C_j^A \in S} C_j^A(\vec{X})]$.

With respect to complexity, there are potential expensive steps in algorithm CSUP$^-$. Projection is required to meet the preconditions of the algorithm and also in step 2. Moreover in the worst case, step 2 of CSUP$^-$ can be exponential in the size of $\mathcal{H}^-$ as it involves distribution of $\wedge$ over $\vee$. In this case, step 4 requires an exponential number of calls to the constraint solver. However, note that the point of this section is to present a decidable case for CSUP.

6.2 Decidability of CSUP for $\mathcal{Q}_{lin}$

In the preceding subsection, we have shown that the assumptions A1 and A2 are sufficient to encode the definition of the CSUP without the groundness condition into the algorithm CSUP$^-$. Now, to define a procedure for solving the CSUP, we need to take advantage of specific properties of the constraint structure. We present such a procedure for CLP($\mathcal{Q}_{lin}$) in a bottom-up approach. First, given a variable X and a satisfiable constraint C, the function FPS finds a value c for X such $X = c \wedge C$ remains satisfiable. Then given a constraint atom A, a set of positive atoms $\mathcal{H}^+$ and a set G of variables to be grounded, the procedure GRND tries to ground G while ensuring that A remains unifiable with each atom of $\mathcal{H}^+$. Finally we present the algorithm CSUP, which iterates calls to GRND to each subspace returned by CSUP$^-$.

Finding a Partial Solution As we do not include disequality into the set of predefined atomic constraints, the space of solutions described by a constraint is convex. Our procedure, see Figure 4, is based on linear programming, hence its complexity is polynomial in theory. The main difficulty is dealing with strict inequalities, e.g., the maximum of X within $X < 0$, denoted $\max_{X<0} X$ is 0, but 0 is *not* a solution for X in $X < 0$. Indeed, the interface predicates for the CLP($\mathcal{Q}$) library described in [7] are called `inf/2` and `sup/2` to emphasize that the minimum (resp. the maximum) of a linear expression over a constraint possibly including strict inequalities is actually an infinum (resp. a supremum). The procedure projects the current constraint C onto the variable X. This projection is a convex set, so even in the presence of strict inequalities, one can easily find a value x_0 for X. By definition of projection, we know that we can extend the partial solution $X = x_0$ to a full solution of C where $X = x_0$. Hence $C \wedge X = x_0$ is satisfiable.

Preconditions: Let X be a variable and C be a satisfiable constraint. There is no condition between X and $\mathcal{V}ar(C)$.

Postcondition: An equation $X = x_0$, where x_0 denotes a rational number, such that $X = x_0 \wedge C$ is satisfiable.

Algorithm:

1. If X is bounded from below within C then
   (a) Let $min := \min_C X_i$
   (b) If X_i is bounded from above within C then
      i. Let $max := \max_C X_i$
      ii. $S := X = (min + max)/2$
   (c) Else $S := X = min + 1$
2. Elif X is bounded from above within C then
   (a) Let $max := \max_C X$
   (b) $S := X = max - 1$
3. Else $S := X = 0$
4. Return S

Fig. 4. Algorithm FPS(X, C).

Theorem 3. *Algorithm FPS terminates and is correct.*

Proof. Termination is obvious. FPS always returns an equation $X = x_0$ where x_0 denotes a rational number. For correctness, let us show that $X = x_0 \wedge C$ is satisfiable. Given the variable X and following the code of the algorithm, there are four cases.

1. X is bounded by min and max.
   (a) If $min = max$, because the linear programming solver is correct, then min is the only possible value for X within the constraint C. Hence $X = min \wedge C$ is satisfiable, and this constraint is equivalent to $X = (min + max)/2 \wedge C$.
   (b) Otherwise, because of the convexity of C, the half sum $(min + max)/2$ is a possible value for X within the constraint C. Hence $X = (min + max)/2 \wedge C$ is satisfiable.
2. X is bounded by min but not bounded from above. Then because of the convexity of C, $min + 1$ is a possible value for X within the constraint C (while we don't know for min). So $X = min + 1 \wedge C$ is satisfiable.
3. X is bounded by max but not bounded from below. Then by convexity $max - 1$ is a possible value for X. The constraint $X = max - 1 \wedge C$ is satisfiable.
4. Otherwise, as C is satisfiable and X is not bounded within C, 0 is a possible value for X, and the constraint $X = 0 \wedge C$ is satisfiable.

So in all cases, the equation returned by the algorithm is indeed a solution for X of the input constraint C.

Grounding In Figure 5, we present an algorithm that grounds some variables while ensuring that the subspace it defines intersects with all positive atoms. First we need the projection of a constraint onto a variable, which can be computed by linear programming.

Definition 5. *Let c be a satisfiable constraint and X be a variable. Then* $\mathrm{proj}(c, X)$ *denotes the constraint $c'(X)$ which is the projection of c onto X.*

Preconditions: Let A be a projected constraint atom $\langle c_A | p(\vec{X})\rangle$, let $\mathcal{H}^+$ be a finite set of projected constraint atoms such that all of them are pairwise variable disjoint and $A \approx B$ for all $B \in \mathcal{H}^+$, and let $\{X_1, \ldots, X_n\} \subseteq \mathcal{V}ar(A)$.

Postcondition: $\bot$ or a conjunction of equations $S \equiv \wedge_{1 \leq i \leq n} X_i = x_i$ such that $\forall H \in \mathcal{H}^+$, $\langle c_A \wedge S | p(\vec{X})\rangle \approx H$.

Algorithm:

1. $i := 1$
2. $S := \top$
3. While $i \leq n$ do
   (a) $P_i := S \wedge c_A \wedge [\wedge_{H \in \mathcal{H}^+} proj(S \wedge c_A \wedge H(\vec{X}), X_i)]$
   (b) If P_i is satisfiable then $S := FPS(X_i, P_i) \wedge S$
   (c) Else Return $\bot$
   (d) $i := i + 1$
4. Return S

Fig. 5. Algorithm GRND$(A, \mathcal{H}^+, \{X_1, \ldots, X_n\})$.

Theorem 4. *Algorithm GRND terminates and is correct and complete.*

Proof.

- Termination is readily checked.
- For correctness, if the algorithm returns $\bot$, the answer is correct. Otherwise, let us show that $I : (i \leq n+1) \bigwedge S \equiv \wedge_{1 \leq j \leq i-1} X_j = x_j \bigwedge \forall H \in \mathcal{H}^+, \langle c_A \wedge S | p(\vec{X})\rangle \approx H$ is an invariant at line 3. The first time we arrive at line 3, $i = 1$, I is true thanks to the preconditions. Assume I is true line 3. P_i is computed and assumed satisfiable. Algorithm FPS returns an equation $X_i = x_i$ such that $P_i \wedge X_i = x_i$ is satisfiable. Hence, $S \wedge c_A \wedge X_i = x_i$ is satisfiable, and because it intersects the projection of each $H \in \mathcal{H}^+$ onto Xi, any partial solution where $X_1 = x_1, \ldots, X_i = x_i$ can be extended into a solution satisfying $c_A \wedge S \wedge X_i = x_i \wedge H(\vec{X})$ for any $H \in \mathcal{H}^+$. Then i is incremented and the invariant holds again. Line 4, we know that $i > n$ and I, hence the postcondition holds.
- For completeness, if the algorithm returns $\bot$ then there does not exist a partial solution S fixing $\{X_1, \ldots, X_n\}$ such that $\forall H \in \mathcal{H}^+$, $\langle c_A \wedge S | p(\vec{X})\rangle \approx H$.

CSUP with the Groundness Condition

Finally, algorithm CSUP defined in Figure 6 solves the constraint selective unification problem in $\mathcal{Q}_{lin}$.

Theorem 5. *Algorithm CSUP terminates. It correctly and completely solves the CSUP problem for CLP($\mathcal{Q}_{lin}$).*

Preconditions: Let A be a projected constraint atom $\langle c_A | p(\vec{X})\rangle$, let $\mathcal{H}^+$ and $\mathcal{H}^-$ be finite sets of projected constraint atoms such that all of them, including A, are pairwise variable disjoint and $A \approx B$ for all $B \in \mathcal{H}^+ \cup \mathcal{H}^-$, and let $G \subseteq \mathcal{V}ar(A)$.

Postcondition: A possibly empty finite set of constraints, each of them being a solution of $\mathcal{P}(A, \mathcal{H}^+, \mathcal{H}^-, G)$.

Algorithm:

1. $S := CSUP^-(A, \mathcal{H}^+, \mathcal{H}^-)$
2. $T := \emptyset$
3. For each $C_j \in S$ do
   (a) $U := GRND(\langle C_j | p(\vec{X})\rangle, \mathcal{H}^+, G)$
   (b) If $U \neq \bot$ then $T := T \cup \{C_j \wedge U\}$
4. Return T

Fig. 6. Algorithm CSUP.

Proof.

- Termination. The call to CSUP$^-$ terminates and computes a finite set of constraints. The loop of line 3 goes along this finite set and each call to GRND terminates. So algorithm CSUP terminates.
- Correctness. If algorithm CSUP returns a non-empty set of constraints, each of them is a solution to $\mathcal{P}(A, \mathcal{H}^+, \mathcal{H}^-, G)$.
- Completeness. If algorithm CSUP returns the empty set, $\emptyset$, then $\mathcal{P}(A, \mathcal{H}^+, \mathcal{H}^-, G)$ has no solution.

Example 13. Let

- $A := \langle 1 \leq X \wedge X \leq 6 \,|\, p(X)\rangle$,
- $\mathcal{H}^+ := \{\langle 2 < X1 \,|\, p(X1)\rangle, \langle X2 \leq 5 \,|\, p(X2)\rangle\}$,
- $\mathcal{H}^- := \{\langle 3 \leq X3 \wedge X3 \leq 4 \,|\, p(X3)\rangle\}$.

We run the CSUP algorithm for different values of G:

- For $G = \emptyset$, we get two solutions: $\{1 \leq X \wedge X < 3 \ ; \ 4 < X \wedge X \leq 6\}$. For instance let us check the first one. It is satisfiable and it results from the conjunction of c_A: $1 \leq X \wedge X \leq 6$ with the constraint $X < 3$. The constraint atom $\langle 1 \leq X \wedge X < 3 \,|\, p(X)\rangle$ is unifiable with the first positive constraint atom of $\mathcal{H}^+$ as $X = X1 \wedge 1 \leq X \wedge X < 3 \wedge 2 < X1$ is satisfiable and similarly for the second positive constraint atom. It is not unifiable with the negative constraint atom of $\mathcal{H}^-$ as $X = X3 \wedge 1 \leq X \wedge X < 3 \wedge 3 \leq X3 \wedge X3 \leq 4$ is unsatisfiable.
- For $G = \{X\}$, we get $\{X = 5/2 \ ; \ X = 9/2\}$. Again we have two solutions and X is ground as required. Let us check for instance the first one. The satisfiable constraint $X = 5/2$ is equivalent to the conjunction of c_A: $1 \leq X \wedge X \leq 6$ with the constraint $X = 5/2$. The constraint atom $\langle X = 5/2 \,|\, p(X)\rangle$ is unifiable with the first positive constraint atom of $\mathcal{H}^+$ as $X = X1 \wedge X = 5/2 \wedge 2 < X1$ is satisfiable and similarly for the second positive constraint atom. On the other hand, it is not unifiable with the negative constraint atom of $\mathcal{H}^-$ as $X = X3 \wedge X = 5/2 \wedge 3 \leq X3 \wedge X3 \leq 4$ is unsatisfiable.

Example 14. Let

- $A := \langle 0 \leq X \wedge 0 \leq Y \,|\, p(X, Y)\rangle$,

– $\mathcal{H}^+ := \left\{ \begin{array}{l} \langle Y1 \leq X1 - 4 \,|\, p(X1, Y1)\rangle, \\ \langle X2 \leq 8 \wedge 8 \leq Y2 \,|\, p(X2, Y2)\rangle \end{array} \right\}$,
– $\mathcal{H}^- := \{\langle Y3 \leq 2 \,|\, p(X3, Y3)\rangle, \langle X4 \leq 4 \,|\, p(X4, Y4)\rangle\}$.

We give a geometrical interpretation of this CSUP, see Figure 7. First, we have considered the first quadrant of the plane (restricted to $X < 15$ and $Y < 15$) as the solutions has to lie inside the space of solutions of the constraint atom A. In green (light grey if in B&W), we have the two positive spaces: $Y \leq X - 4$ in the lower right and $X \leq 8 \wedge 8 \leq Y$ in the upper left. Above the positive spaces, we have the two negative spaces $Y \leq 2$ and $X \leq 4$ in red (dark grey if in B&W). We run the CSUP algorithm for different values of G:

– For $G = \emptyset$, we get $\{4 < X \wedge 2 < Y\}$. Let us check this solution geometrically. This space is the union of the two green areas with the white one in between. It lies inside c_A: $0 \leq X \wedge 0 \leq Y$, has a non-empty intersection with the two positive spaces and an empty intersection with the negative spaces.
– For $G = \{Y\}$, we get $\{4 < X \wedge Y = 9\}$. Let us check that the half-line $4 < X \wedge Y = 9$, drawn in blue (black if B&W), is indeed a solution. We note that Y is ground as required. The half-line has a non-empty intersection with both positive spaces and an empty intersection with the negative spaces. Moreover, the half-line is included into the first quadrant.
– For $G = \{X\}$, we get $\{X = 7 \wedge 2 < Y\}$. A similar reasoning shows that this half-line is indeed a solution. It intersects the positive spaces, does not intersect the negative spaces and lies inside the first quadrant.
– For $G = \{X, Y\}$, there is no solution. Geometrically, $G = \{X, Y\}$ means: can one find a point which belongs to the green upper left space and at the same time to the green lower right space? No, as the two spaces are disjoint. Hence there is no solution.

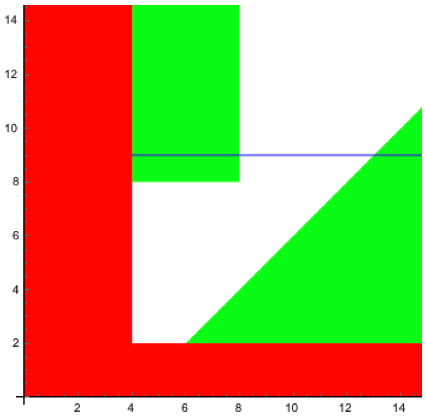

Fig. 7. A graphical representation of a CSUP.

7 Related Work and Conclusion

Constructive negation in LP [2] and, more specifically, in CLP [20] is related to our work. The starting point of constructive negation stems from the desire to extract constructive information from the proof of a negative subgoal, in contrast to the usual negation as failure rule. This is precisely what we do in algorithm CSUP$^-$ while processing the negative atoms of $\mathcal{H}^-$. In [20], Stuckey introduced *admissible closedness* as a sufficient condition over constraint structures for a practical use of constructive negation. This property is weaker than quantifier elimination. Stuckey showed that assumptions A1 and A2 of section 6 imply admissible closedness. Moreover CLP($\mathcal{H}$), the constraint domain of finite trees with equality and quantified disequality does not admit quantifier elimination but is admissible closed. More recently, Dovier et al. proved that admissible closedness was also a necessary condition for constructive negation in CLP [4]. So this concept could be a promising tool for studying concolic testing for logic programming, as we have begun in [16, 17] but this time from a CLP perspective.

Other future work could investigate the idea of grabing specific values from the concrete component of concolic execution in order to help solving difficult constraints. For instance, $\mathcal{Q}_{lin}$ cannot deal with non-linear constraints that can appear in the symbolic part of concolic testing. So linearizing such constraints by introducing concrete data is an interesting possibility.

A third idea that seems appealing is the inclusion of interpolation. It has been previously shown that interpolation [3] can improve concolic testing [12] in an imperative setting. CLP evaluation [13, 6] can also benefit from interpolation. It remains to see if one can combine both approaches.

Summarizing this paper, we have considered concolic testing in the framework of constraint logic programming. We have proved that the selective unification problem is generally undecidable for CLP. For a restricted class of constraint structures, we have given a generic correct and complete algorithm for selective unification without the groundness condition. Finally, we have presented a specific selective unification with the groundness condition for CLP($\mathcal{Q}_{lin}$).

References

1. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In E. A. Emerson and K. S. Namjoshi, editors, *Proc. of VMCAI'06*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.
2. D. Chan. Constructive negation based on the completed database. In *ICLP/SLP*, pages 111–125. MIT Press, 1988.
3. W. Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *J. Symb. Log.*, 22(3):269–285, 1957.
4. A. Dovier, E. Pontelli, and G. Rossi. A necessary condition for constructive negation in constraint logic programming. *Inf. Process. Lett.*, 74(3-4):147–156, 2000.
5. John P. Gallagher and Maurice Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Comput.*, 9(3/4):305–334, 1991.
6. G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Failure tabled constraint logic programming by interpolation. *TPLP*, 13(4-5):593–607, 2013.
7. C. Holzbaur. OFAI clp(q,r) manual, edition 1.3.3. Technical Report TR-95-09, Austrian Research Institute, 1995.
8. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 1979.
9. J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. of 14th Annual ACM Symp. on Principles of Programming Languages*, pages 111–119. ACM, 1987.

10. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19,20:503–581, 1994.
11. J. Jaffar, M. J. Maher, K. Marriott, and P. J. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1-3):1–46, 1998.
12. J. Jaffar, V. Murali, and J. A. Navas. Boosting concolic testing via interpolation. In *ESEC/SIGSOFT FSE*, pages 48–58. ACM, 2013.
13. J. Jaffar, A. E. Santosa, and R. Voicu. An interpolation method for CLP traversal. In *CP*, volume 5732 of *Lecture Notes in Computer Science*, pages 454–469. Springer, 2009.
14. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. 2nd Ed.
15. K. Marriott and P. J. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, Cambridge, MA, 1998.
16. F. Mesnard, É. Payet, and G. Vidal. Concolic testing in logic programming. *TPLP*, 15(4-5):711–725, 2015.
17. F. Mesnard, É. Payet, and G. Vidal. On the completeness of selective unification in concolic testing of logic programs. *CoRR*, abs/1608.03054, 2016.
18. P. Refalo and P. Van Hentenryck. CLP($\mathcal{R}_{lin}$) revised. In M. Maher, editor, *Proc. of the Joint International Conf. and Symposium on Logic Programming*, pages 22–36. The MIT Press, 1996.
19. T. Ströder, F. Emmes, P. Schneider-Kamp, J. Giesl, and C. Fuhs. A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog. In *LOPSTR'11*, pages 237–252. Springer LNCS 7225, 2011.
20. P. J. Stuckey. Negation and constraint logic programming. *Inf. Comput.*, 118(1):12–33, 1995.