LSign Reordered

Viswanath Ramachandran and Pascal Van Hentenryck

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-95-12
April 1995

LSign Reordered

Viswanath Ramachandran and Pascal Van Hentenryck
Brown University, Box 1910, Providence, RI 02912 (USA)
{vr,pvh}@cs.brown.edu

Abstract

LSign [18] is an elegant domain for analyzing constraint logic programming languages over linear real constraints. Its key conceptual ideas are the abstraction of linear constraints by replacing coefficients by their signs and the use of annotations for preserving multiplicity information on the constraints. Unfortunately, the ordering of abstract constraint stores given in [18] does not capture the intended meaning and makes it impossible to prove the consistency of the abstract operations of LSign. In this paper, we reconsider the domain LSign. Our first contribution is to define a new ordering on abstract constraint stores and to give the meaning of the domain through a monotone concretization function from abstract stores to sets of constraint stores, where a constraint store is a multiset of constraints. Our second contribution is to show that ordering two abstract stores reduces to a matching problem, which can be solved in polynomial time. Our last contribution is to propose a simpler and more precise algorithm for abstract projection and a schema for generating upper bound operations and to prove their consistency.

1 Introduction

Constraint logic programming (CLP) [10, 4] is a generalization of logic programming where unification is replaced by constraint solving over some computation domain. Many CLP languages have been defined in the last decade on computation domains such linear real constraints (e.g., [11, 22, 4]), integers (e.g., [21]), Booleans (e.g., [3, 4]), and nonlinear real constraints (e.g., [20]) to name a few. Recently, more and more attention has been devoted to abstract interpretation of constraint logic programming (e.g., [16, 9]) and of CLP($\Re_{lin}$), an instance of CLP over linear real constraints, in particular (e.g., [17, 18, 7]). This interest is motivated by the large potential for optimization in CLP($\Re_{lin}$) programs using techniques such as refinements, reordering, and removal of constraints [17].

Reordering constraints is a particularly interesting avenue since it enables other optimizations to be applied more effectively. Marriott and Stuckey [17] have shown that determining when reordering can be performed reduces to checking if a constraint store is always satisfiable. A variety of abstract domains can be used for this task, including the abstract domain of Cousot and Halbwachs [6] and the LSign domain of Marriott and Stuckey [18]. LSign is a new and elegant domain whose key conceptual ideas are the abstraction of linear constraints by replacing coefficients by their signs and the use of annotations for preserving multiplicity information on the constraints. This allows the projection operation to be approximated by abstract versions of Fourier and Gauss elimination.

Unfortunately, the ordering given in [18], i.e.,

$$\beta_1 \sqsubseteq \beta_2 \Leftrightarrow \forall \gamma_1 \in \beta_1 \ \exists \gamma_2 \in \beta_2 \quad \gamma_1 \sqsubseteq \gamma_2$$

does not capture the intended meaning, since it would conclude that the set $\beta_1 = \{\oplus = \oplus x_1 + \oplus x_2, \oplus = \oplus x_1 + \ominus x_2\}$ is smaller than $\beta_2 = \{\oplus = \oplus x_1 + \top x_2\}$, where $\oplus$ represents a strictly positive coefficient, $\ominus$ a strictly negative coefficient, and $\top$ any coefficient. The intended meaning of [18] is that β_1 represents exactly two constraints while β_2 represents exactly one constraint. Hence β_2 is always satisfiable, while β_1 may not be. We observed this problem when trying to define an upper bound operation for `LSign` which was not given in [18]. Obviously, if two distinct clauses return $\{\oplus = \oplus x_1 + \oplus x_2\}$ and $\{\oplus = \oplus x_1 + \ominus x_2\}$, we were tempted to return $\{\oplus = \oplus x_1 + \top x_2\}$ as an upper bound but noticed that $\beta_1 = \{\oplus = \oplus x_1 + \oplus x_2, \oplus = \oplus x_1 + \ominus x_2\}$ was smaller wrt the defined ordering. This ordering problem obviously makes it impossible to prove the correctness of the abstract operations of `LSign`.

The purpose of this paper is to reconsider the domain `LSign` and its abstract operations. Our starting point is a slight generalization of `LSign`, which clearly separates multiplicity information from the abstract constraints.[1] Our first contribution is to define an ordering on this domain and to capture the intended meaning of the domain through a monotone concretization function from abstract constraint stores to sets of constraint stores, where a constraint store is a multiset of constraints.[2] Our second contribution is to show that ordering two abstract constraint stores reduces to a matching problem, which can be solved in polynomial time. Our last contribution is to reconsider the other abstract operations of `LSign`, i.e., projection, upper bound, and addition of a constraint, and to prove their consistency. In particular, we propose a simpler and more precise algorithm for abstract projection which factorizes the case analysis in a single procedure and a schema for generating consistent upper bound operations. It is important to stress that our goal in this paper is not to improve the `LSign` domain but rather to show how the results of [18] can be corrected and completed, since they seem to be particularly appropriate for optimizing compilers of `CLP`($\Re_{lin}$).

The rest of this paper is organized in the following way. Section 2 gives the background for this paper. It contains an overview of `CLP`($\Re_{lin}$) programs, a brief summary of our approach to abstract interpretation and the definition of the concrete objects. Section 3 contains a description of the abstract domain `LSign`, including the concretization function and its proof of monotonicity. Section 4 contains the implementation of the abstract operations (i.e., ordering, addition of a constraint, upper bound, projection) and their consistency proofs. Section 5 contains the conclusion of the paper.

2 Preliminaries

`CLP`($\Re_{lin}$) **Programs** As in [18], we consider the analysis of CLP programs containing only constraints in $\Re_{lin}$ (i.e. no functor symbol). To define the syntax of `CLP`($\Re_{lin}$) programs precisely, we assume the existence of sets P_i ($i \geq 0$) denoting sets of predicate symbols of arity i and of an infinite set PV of program variables. Variables in PV are ordered and denoted by $x_1, x_2, \ldots, x_i, \ldots$.

`CLP`($\Re_{lin}$) programs consists of clauses of the form $p(x_1, \ldots, x_n) :- g_1, \ldots, g_l$ where $n \geq 0$, $p \in P_n$, and $l \geq 0$. Each goal g_i in a clause is an atom of the form $p(x_{i_1}, \ldots, x_{i_k})$ where $x_{i_1}, \ldots, x_{i_k}$ are all distinct program variables and $p \in P_k$ or a constraint of the form $c_0 \; op \; \sum_{i=1}^{m} c_i x_i$ where c_i are rational numbers, $op \in \{<, \leq, =\}$, and $x_1, \ldots, x_m$ are all the clause variables. A typical `CLP`($\Re_{lin}$) program is the factorial program

[1] This was motivated by our previous work on sequences [12, 1] which separates properties of the elements of the sequences from properties of the sequence.

[2] Note that Marriott and Stuckey use an abstraction function for constraints and an approximation relation for the remaining semantic objects.

```
fact(x1,x2) :- x1 = 0, x2 = 1.
fact(x1,x2) :- x1 > 0, x3 = x1 − 1, x2 = x4 × x1, fact(x3,x4).
```

Overview of the Framework Our abstract interpretation is a natural extension to CLP of our logic programming framework [14]. It is thus close to the work of Marriott and Søndergaard [15, 16], Winsborough [23]. Also related are the frameworks of Bruynooghe [2] and Hermenegildo [19, 9]. Our framework follows the traditional approach to abstract interpretation [5].

As is traditional in abstract interpretation, the starting point of the analysis is a collecting semantics for the programming language. Our concrete semantics is a collecting fixpoint semantics which captures the top-down execution of constraint logic programs using a left-to-right computation rule and which ignores the clause selection rule. The semantics manipulates sets of constraint stores, i.e. multisets of linear constraints. Two main operations are performed on constraint stores: addition of a constraint to a constraint store and projection of a variable from a constraint store. The semantics associates with each of the predicate symbol p in the program a set of tuples of the form $(\Theta_{in}, p, \Theta_{out})$ which can be interpreted as follows:

> "the execution of $p(x_1, \ldots, x_n) \ \wedge\ \theta$ with $\theta \in \Theta_{in}$ produces a set of constraint stores $\{\theta_1, \ldots, \theta_n, \ldots\}$, all of which belongs to Θ_{out}."

The second step of the methodology is the abstraction of the concrete semantics. Our abstract semantics consists in abstracting a set of constraint stores by a single abstract store, i.e. an abstract store represents a set of constraint stores. As a consequence, the abstract semantics associates with each predicate symbol p a set of tuples of the form $(\beta_{in}, p, \beta_{out})$ which can be read informally as follows:

> "the execution of $p(x_1, \ldots, x_n) \ \wedge\ \theta$ with θ satisfying the property described by β_{in} produces a set of constraint stores $\theta_1, \ldots, \theta_n, \ldots$, all of which satisfy the property β_{out}."

In our approach, the link between the abstract and the concrete domain is given by a monotone concretization function. The abstract semantics assumes a number of operations on abstract stores, in particular addition of a constraint, projection, and upper bound. The first two operations are simply consistent approximations of the corresponding concrete operations. The upper bound operation is a consistent abstraction of union of sets of constraint stores. *The monotonicity of the concretization function and the consistency of the abstract operations guarantee that any postfixpoint of the abstract transformation is an approximation of the least fixpoint of the concrete transformation* (see [14] for the details).

The last step of the methodology consists in computing the least fixpoint or a postfixpoint of the abstract semantics. `GAIA` [14] is a top-down algorithm computing a small, but sufficient, subset of least fixpoint (or of a postfixpoint) necessary to answer a user query. The algorithm uses memoization, a dependency graph to avoid redundant computation, the abstract operations of the abstract semantics, and the ordering relation on the abstract domain. It has many similarities with `PLAI` [19] and can be seen either as an implementation of Bruynooghe's framework [2] or as an instance of a general fixpoint algorithm [13].

Concrete Objects The concrete objects manipulated by our concrete semantics are linear constraints and multisets of linear constraints. Consider a set of variables $D = \{x_1, \ldots, x_n\}$. A linear constraint over D is an expression of the form $c_0 \ op \ \sum_{i=1}^{n} c_i x_i$ where c_i are rational numbers and $op \in \{<, \leq, =\}$. The set of linear constraints over D is denoted by C_D. A constraint store over D is simply a multiset of linear constraints over D. The set of constraint stores over D is denoted by CS_D and is ordered by standard multiset inclusion. We denote multiset union by $\uplus$. In the following, we denote linear constraints by the letter λ and constraint stores by the letter θ, both possibly subscripted. If λ is a linear constraint $c_0 \ op \ \sum_{i=1}^{n} c_i x_i$, $\lambda[i]$ denotes the coefficient c_i and $op(\lambda)$ denotes the operator op. If θ is a constraint store over $D = \{x_1, \ldots, x_n\}$ and $x \in D$, θ_x^+, θ_x^-, θ_x^0, represents all constraints λ in θ whose coefficient for variable x is respectively strictly positive, strictly negative, and zero.

3 The Abstract Domain LSign

In this section, we introduce the domain LSign. Although the presentation differs considerably from [18], the domain is in fact a slight generalization of the original domain. The new presentation is motivated by the fact that it makes it easy to define the concretization function compositionally by identifying the semantic objects clearly. In contrast, [18] uses an approach based on an abstraction function and approximation relations. The first key idea is the notion of an abstract constraint which abstracts a concrete constraint by replacing each coefficient by its sign. Our definitions assume $D = \{x_1, \ldots, x_n\}$.

Definition 1 [Signs] A *sign* is an element of $\mathtt{Sign} = \{0, \oplus, \ominus, \top\}$. Sign is ordered by

$$s_1 \sqsubseteq s_2 \Leftrightarrow (s_1 = s_2) \vee (s_2 = \top)$$

The monotone concretization function $Cc : Sign \to 2^{\Re}$ is defined as

$$Cc(0) = \{0\}; \quad Cc(\oplus) = \{c \mid c \in \Re \wedge c > 0\}; \quad Cc(\ominus) = \{c \mid c \in \Re \wedge c < 0\}; \quad Cc(\top) = \Re.$$

It is clear that the Sign domain can be extended easily to cover $\Re \setminus \{0\}$, $\Re^+$, and $\Re^-$.

Definition 2 [Operators] An *operator* is an element of $\mathtt{Op} = \{<, \leq, =\}$. Op is ordered by

$$op_1 \sqsubseteq op_2 \Leftrightarrow (op_1 = op_2)$$

Definition 3 [Abstract Constraints] An *abstract constraint* over D is an expression of the form $s_0 \ op \ \sum_{i=1}^{n} s_i x_i$ where s_i is a sign and op is an operator. The set of abstract constraints over D is denoted by A_D and is ordered by

$$s_0 \ op \ \sum_{i=1}^{n} s_i x_i \sqsubseteq s_0' \ op' \ \sum_{i=1}^{n} s_i' x_i' \Leftrightarrow (op \sqsubseteq op') \wedge \bigwedge_{i=0}^{n} (s_i \sqsubseteq s_i').$$

The concretization function $Cc : A_D \to C_D$ is defined as

$$Cc(s_0 \ op \ \textstyle\sum_{i=1}^{n} s_i x_i) = \{ \ c_0 \ op \ \sum_{i=1}^{n} c_i x_i \mid c_i \in Cc(s_i) \ (1 \leq i \leq n) \ \}.$$

Abstract constraints are denoted by the letter σ, possibly subscripted.

Example 4 The abstract constraint $\oplus = \ominus x_1 + \top x_2$ represents both the constraint $3 = -x_1 + x_2$ and $3 = -x_1 - x_2$ but not the constraint $3 = x_1 - x_2$.

Lemma 5 If σ_1 and σ_2 are abstract constraints then $\sigma_1 \sqsubseteq \sigma_2 \Rightarrow Cc(\sigma_1) \subseteq Cc(\sigma_2)$.

The second key concept is the notion of an abstract constraint with multiplicity which represents a multiset of constraints. The multiplicity information specifies the size of the multiset. We consider three multiplicities, *One*, *ZeroOrOne*, and *Any*, which are used respectively to represent a multiset of size 1, a multiset of size 0 or 1, or a multiset of arbitrary size.[3]

Definition 6 [Multiplicities] A *multiplicity* is an element of `Mult` $= \{One, ZeroOrOne, Any\}$. `Mult` is ordered by $One \sqsubseteq ZeroOrOne \sqsubseteq Any$. Multiplicities are denoted by the letter μ, possibly subscripted.

We now turn to abstract constraint with multiplicities. Recall that elements of CS_D are multisets of linear constraints.

Definition 7 [Abstract Constraints with Multiplicity] An *abstract constraint with multiplicity* over D is an expression of the form $\langle\sigma, \mu\rangle$, where σ is an abstract constraint over D and μ is a multiplicity. The set of abstract constraints with multiplicity over D is denoted by AM_D and is ordered by

$$\langle\sigma_1, \mu_1\rangle \sqsubseteq \langle\sigma_2, \mu_2\rangle \Leftrightarrow \sigma_1 \sqsubseteq \sigma_2 \wedge \mu_1 \sqsubseteq \mu_2.$$

The concretization function $Cc : AM_D \rightarrow CS_D$ is defined as

$$\begin{array}{l} Cc(\langle\sigma, One\rangle) = \{\ \{\lambda\} \mid \lambda \in Cc(\sigma)\ \} \\ Cc(\langle\sigma, ZeroOrOne\rangle) = \{\emptyset\} \cup \{\ \{\lambda\} \mid \lambda \in Cc(\sigma)\ \} \\ Cc(\langle\sigma, Any\rangle) = \{\emptyset\} \cup \{\ \{\lambda_1, \ldots, \lambda_m\} \mid m \geq 1 \wedge \lambda_i \in Cc(\sigma)\ (1 \leq i \leq m)\ \}. \end{array}$$

Abstract constraints with multiplicities are denoted by the letter γ, possibly subscripted. Moreover, if γ is $\langle\sigma, \mu\rangle$, $cons(\gamma)$ denotes σ and $mult(\gamma)$ denotes μ.

Example 8 The abstract constraint with multiplicity $\langle \oplus = \ominus x_1 + \top x_2, One\rangle$ represents only multisets of size 1, e.g., $\{3 = -x_1 + x_2\}$. $\langle \oplus = \ominus x_1 + \top x_2, Any\rangle$ represents multisets of any size, e.g., $\emptyset$, $\{3 = -x_1 + x_2\}$ and $\{3 = -x_1 + x_2, 3 = -x_1 - x_2\}$.

Lemma 9 If γ_1 and γ_2 are abstract constraints with multiplicity then $\gamma_1 \sqsubseteq \gamma_2 \Rightarrow Cc(\gamma_1) \subseteq Cc(\gamma_2)$.

We are now in position to define the abstract stores of the domain.

Definition 10 [Abstract Stores] An *abstract store* over D is a set β of abstract constraints with multiplicities such that

$$\forall \gamma_1, \gamma_2 \in \beta :\ mult(\gamma_1) \in \{ZeroOrOne, Any\} \wedge mult(\gamma_2) = Any\ \wedge\ \gamma_1 \sqsubseteq \gamma_2 \Rightarrow \gamma_1 = \gamma_2.$$

The set of abstract stores is denoted by AS_D. The concretization function $Cc : AS_D \rightarrow CS_D$ is defined as

$$\begin{array}{l} Cc(\emptyset) = \{\emptyset\} \\ Cc(\{\gamma\} \cup \beta) = \{\ \theta_1 \uplus \theta_2 \mid \theta_1 \in Cc(\gamma) \wedge \theta_2 \in Cc(\beta)\ \} \quad \gamma \notin \beta. \end{array}$$

Abstract stores are denoted by the letter β, possibly subscripted.

[3][18] contains one other multiplicity *OneOrMore* which is obtained easily in our domain by including one constraint with multiplicity *One* and one constraint with multiplicity *Any*. Note also that all inequalities are defined with a multiplicity *Any* in [18].

The additional condition in the above definition is a simple subsumption condition stating that a constraint with multiplicity *ZeroOrOne* is not subsumed by a constraint with multiplicity *Any* and that a constraint with multiplicity *Any* is not subsumed by another constraint with multiplicity *Any*. This condition is necessary to obtain a partial order. It should be clear however that removing these constraints does not change the concretization of the abstract store.

Example 11 The abstract store $\{\langle \oplus = \ominus x_1 + \top x_2, One\rangle\ ,\ \langle \oplus = \ominus x_1 + \oplus x_2, Any\rangle\}$ represents constraint stores with at least one constraint, e.g., $\{3 = -x_1 + x_2\ ,\ 2 = -x_1 + 3x_2\}$.

It remains to define the ordering on abstract stores. Our goal is to make sure that $\beta_1 \leq \beta_2$ implies that $Cc(\beta_1) \subseteq Cc(\beta_2)$ to obtain a monotone concretization function. The ordering relation is non-trivial and requires the following concepts.

Definition 12 [Definite, Possible, and Indefinite Constraints] Let γ be an abstract constraint with multiplicity. γ is a *definite* abstract constraint iff $mult(\gamma) = One$. γ is a *possible* abstract constraint iff $mult(\gamma) = ZeroOrOne$. It is an *indefinite* abstract constraint otherwise. The *definite portion* of β, denoted by $\texttt{Def}(\beta)$, is the set of definite constraints of β. The *possible portion* of β, denoted by $\texttt{Pos}(\beta)$, is the set of possible constraints of β. The *indefinite portion* of β, denoted by $\texttt{Indef}(\beta)$, is the set of indefinite constraints of β.

Definition 13 [Ordering Function] Let β_1 and β_2 be two abstract stores. An *ordering function* of β_1 to β_2 is a function $f : \beta_1 \rightarrow \beta_2$ satisfying

1. $\forall\ \gamma \in \beta_1$: $\gamma \sqsubseteq f(\gamma)$.
2. $\forall\ \gamma_1, \gamma_2 \in \beta_1$: $\gamma_1 \neq \gamma_2 \Rightarrow f(\gamma_1) \neq f(\gamma_2) \vee f(\gamma_1) \in \texttt{Indef}(\beta_2)$.
3. $\texttt{Def}(\beta_2) \subseteq codomain(f)$.

Definition 14 [Ordering on Abstract Stores] Let β_1 and β_2 be two abstract stores. $\beta_1 \sqsubseteq \beta_2$ if and only if there exists an ordering function f of β_1 to β_2.

The first condition simply states that, for each abstract constraint with multiplicity in β_1, say γ, there exists an abstract constraint with multiplicity in β_2, i.e., $f(\gamma)$, that approximates it. The next two conditions concern the number of constraints. The second condition requires that each definite constraint and each possible constraint in β_2 be used at most once. The third condition requires that each definite constraint in β_2 be used at least once.

Example 15 Consider the following abstract constraints with multiplicity.

$$\begin{aligned}
\gamma_1 &= \langle \oplus = \oplus x_1 + \oplus x_2, One\rangle \\
\gamma_2 &= \langle \oplus = \ominus x_1 + \oplus x_2, One\rangle \\
\gamma_3 &= \langle \oplus = \ominus x_1 + \oplus x_2, Any\rangle \\
\gamma_4 &= \langle \oplus = \oplus x_1 + \oplus x_2, One\rangle \\
\gamma_5 &= \langle \oplus = \ominus x_1 + \top x_2, Any\rangle \\
\gamma_6 &= \langle \oplus = \oplus x_1 + \top x_2, One\rangle
\end{aligned}$$

$\{\gamma_1, \gamma_2\} \sqsubseteq \{\gamma_4, \gamma_5\}$ and the ordering function is defined by $f(\gamma_1) = \gamma_4$ and $f(\gamma_2) = \gamma_5$.
$\{\gamma_1, \gamma_2, \gamma_3\} \sqsubseteq \{\gamma_4, \gamma_5\}$ and the function is defined by $f(\gamma_1) = \gamma_4$, $f(\gamma_2) = \gamma_5$, and $f(\gamma_3) = \gamma_5$.
It is not true that $\{\gamma_3, \gamma_4\} \sqsubseteq \{\gamma_1, \gamma_5, \gamma_6\}$, since there is no function that can cover both γ_1 and γ_6 with only $\{\gamma_3, \gamma_4\}$.

We show that $\sqsubseteq$ on abstract stores is a partial order.

Theorem 16 [Ordering Relation] $\sqsubseteq$: $AS_D \times AS_D$ is a partial order.
Proof See Appendix A. $\Box$

We now turn to the first main result of this paper: the monotonicity of the concretization function. The proof uses several lemmas, one of them (i.e., the lifting function lemma) being fundamental in all consistency proofs. Note that we sometimes abuse notation by writing expressions like $\lambda_1 \neq \lambda_2$ to mean that λ_1 and λ_2 are two different constraints or two different occurrences of the same constraint in a multiset. These abuses should be clear from the context. We also write $|S|$ to denote the cardinality of a set or of a multiset.

Lemma 17 Let γ be an abstract constraint with multiplicity and θ a concrete constraint store. $\theta \in Cc(\gamma) \Rightarrow \forall \lambda \in \theta : \lambda \in Cc(cons(\gamma))$.
Proof Immediate consequence of Definition 7 $\Box$

We define to the notion of lifting function which is the counterpart of the ordering function for a pair (concrete store, abstract store).

Definition 18 [Lifting Function] Let β be an abstract store and θ a concrete constraint store. A *lifting function* of θ to β is a function $f : \theta \to \beta$ satisfying

1. $\forall \lambda \in \theta$: $\lambda \in Cc(cons(f(\lambda)))$.
2. $\forall \lambda_1, \lambda_2 \in \theta$: $\lambda_1 \neq \lambda_2 \Rightarrow f(\lambda_1) \neq f(\lambda_2) \vee f(\lambda_1) \in \mathtt{Indef}(\beta)$.
3. $\mathtt{Def}(\beta) \subseteq codomain(f)$.

As mentioned, the following lemma is the cornerstone of most proofs in this paper.

Lemma 19 [Lifting Function Lemma] Let β be an abstract store and θ a concrete constraint store. $\theta \in Cc(\beta)$ if and only if there exists a lifting function f of θ to β.
Proof See Appendix A. $\Box$

The following result is a simple consequence of the definition of the orderings.

Lemma 20 Let γ_1 and γ_2 be two abstract constraints with multiplicity and let $\gamma_1 \sqsubseteq \gamma_2$. If γ_2 is a definite constraint, then γ_1 is also a definite constraint. If γ_1 is an indefinite constraint, then γ_2 is also an indefinite constraint.

We are now in position to prove the first main result of this paper.

Theorem 21 [Monotonicity of Concretization Function w.r.t. Ordering Relation] If β_1 and β_2 are two abstract stores then $\beta_1 \sqsubseteq \beta_2 \Rightarrow Cc(\beta_1) \subseteq Cc(\beta_2)$.
Proof Let $\beta_1 \sqsubseteq \beta_2$ and $\theta \in Cc(\beta_1)$. We show that $\theta \in Cc(\beta_2)$. By Lemma 19, there exists a lifting function f of θ to β_1. By Definition 14 there exists an ordering function g of β_1 to β_2. Consider the function $g \circ f$. We show that $g \circ f$ is a lifting function of θ to β_2.

1. $\forall \lambda \in \theta : \lambda \in Cc(cons(f(\lambda)))$ by Definition 18
$\forall f(\lambda) \in \beta_1 : f(\lambda) \sqsubseteq g(f(\lambda))$ by Definition 13
$\forall f(\lambda) \in \beta_1 : cons(f(\lambda)) \sqsubseteq cons(g(f(\lambda)))$ by ordering on AM_D
$\forall \lambda \in \theta : \lambda \in Cc(cons(g(f(\lambda))))$ by Lemma 5.

2. $\forall \lambda_1, \lambda_2 \in \theta : \lambda_1 \neq \lambda_2$
$\Rightarrow f(\lambda_1) \neq f(\lambda_2) \vee f(\lambda_1) \in \mathtt{Indef}(\beta_1)$ by Definition 18
$\Rightarrow f(\lambda_1) \neq f(\lambda_2) \vee g(f(\lambda_1)) \in \mathtt{Indef}(\beta_2)$ by $f(\lambda_1) \sqsubseteq g(f(\lambda_1))$ and Lemma 20
$\Rightarrow g(f(\lambda_1)) \neq g(f(\lambda_2) \vee g(f(\lambda_1)) \in \mathtt{Indef}(\beta_2)) \vee g(f(\lambda_1)) \in \mathtt{Indef}(\beta_2)$ by Definition 13
$\Rightarrow g(f(\lambda_1)) \neq g(f(\lambda_2)) \vee g(f(\lambda_1)) \in \mathtt{Indef}(\beta_2)$

3. $\mathtt{Def}(\beta_2) \subseteq codomain(g)$ by Definition 13
$\Rightarrow \forall \gamma_2 \in \mathtt{Def}(\beta_2)\ \exists \gamma_1 \in \beta_1 :\ g(\gamma_1) = \gamma_2$
$\Rightarrow \forall \gamma_2 \in \mathtt{Def}(\beta_2)\ \exists \gamma_1 \in \mathtt{Def}(\beta_1) :\ g(\gamma_1) = \gamma_2$ $\gamma_1 \sqsubseteq g(\gamma_1) = \gamma_2$ and Lemma 20
$\forall \gamma_1 \in \mathtt{Def}(\beta_1)\ \exists \lambda \in \theta :\ f(\lambda) = \gamma_1$ by Definition 18
$\Rightarrow \forall \gamma_2 \in \mathtt{Def}(\beta_2)\ \exists \lambda \in \theta :\ g(f(\lambda)) = \gamma_2$ combining the above two statements
$\Rightarrow \mathtt{Def}(\beta_2) \subseteq codomain(g \circ f)$

$\theta \in Cc(\beta_2)$ follows by Lemma 19. □

4 Implementation of the Abstract Operations

We now study the implementation of the abstract operations of LSign. We start by the implementation of the ordering relation, continue with the addition of a constraint and the upper bound operation and conclude with projection. Note that, in the implementations, we do not remove the subsumed constraints (required by the definition of abstract store) for simplicity. It is easy to apply a postprocessing step to remove them and the correctness obviously follows from the fact that the concretization is not changed by the removal of these constraints.

4.1 Ordering

The problem of ordering abstract stores, i.e. of deciding whether $\beta_1 \sqsubseteq \beta_2$ could be solved simply by enumerating all functions from β_1 to β_2 to determine whether one of them is an ordering function. However, this would lead to an exponential algorithm in the size of the stores. In this section, we show that we can do much better by reducing the ordering problem to a maximum weighted bipartite graph matching problem.

Definition 22 [Bipartite Graph] A graph $G = (V, E)$ is bipartite if V can be partitioned into two sets V_1 and V_2 such that $(u, v) \in E$ implies either $u \in V_1 \wedge v \in V_2$ or $u \in V_2 \wedge v \in V_1$.

Definition 23 [Maximum Weighted Bipartite Graph Matching Problem] Let $G = (V, E)$ be a weighted bipartite graph. A matching M on G is a set of edges no two of which have a common vertex. The weight of M is the sum of its edge weights. The *maximum weighted bipartite graph matching problem* is that of finding a matching of maximum weight.

The key ideas behind the reduction are as follows. The first set of vertices corresponds to the constraints of β_1. The second set of vertices contains a vertex for each constraint in $\texttt{Def}(\beta_2)$, a vertex for each constraint in $\texttt{Pos}(\beta_2)$, and $|\beta_1|$ vertices for each constraint in $\texttt{Indef}(\beta_2)$, since these constraints can be used several times and the matching problem requires that each vertex appears atmost once in a solution. The edges connect vertices from the first set to vertices of the second set only if the constraint in the first set is smaller or equal to the constraint in the second set. This requirement makes sure that the first property of ordering is guaranteed. To ensure the third property, we specify the weights in a special way to encourage the covering of the definite constraints in β_2. The second property will follow from the definition of a matching.

Definition 24 Let β_1 and β_2 be two abstract stores. The *matching graph* of β_1 to β_2 is a (weighted bipartite) graph $G = (V, E)$ such that

$$\begin{array}{lcl}
V & = & V_1 \cup V_2 \cup V_3 \cup V_4 \\
V_1 & = & \{\gamma_1 \mid \gamma_1 \in \beta_1\} \\
V_2 & = & \{\gamma_2 \mid \gamma_2 \in \texttt{Def}(\beta_2)\} \\
V_3 & = & \{\gamma_2 \mid \gamma_2 \in \texttt{Pos}(\beta_2)\} \\
V_4 & = & \{\gamma_2{}^{\gamma} \mid \gamma_2 \in \texttt{Indef}(\beta_2) \wedge \gamma \in \beta_1\} \\
 & & \\
E & = & E_2 \cup E_3 \cup E_4 \\
E_2 & = & \{(\gamma_1, \gamma_2) \mid \gamma_1 \in V_1 \wedge \gamma_2 \in V_2 \wedge \gamma_1 \sqsubseteq \gamma_2\} \\
E_3 & = & \{(\gamma_1, \gamma_2) \mid \gamma_1 \in V_1 \wedge \gamma_2 \in V_3 \wedge \gamma_1 \sqsubseteq \gamma_2\} \\
E_4 & = & \{(\gamma_1, \gamma_2{}^{\gamma_1}) \mid \gamma_1 \in V_1 \wedge \gamma_2{}^{\gamma_1} \in V_4 \wedge \gamma_1 \sqsubseteq \gamma_2\} \\
 & & \\
weight(e) & = & \left\{ \begin{array}{ll} 2 & \text{if } e \in E_2 \\ 1 & \text{if } e \in E_3 \cup E_4 \end{array} \right. \qquad \forall e \in E
\end{array}$$

Implementation 25 [Ordering] Let β_1 and β_2 be two abstract stores. Let G be the matching graph β_1 to β_2. $\beta_1 \sqsubseteq \beta_2$ returns *true* iff the maximum matching of G has weight $|\beta_1| + |\texttt{Def}(\beta_2)|$.

We now prove the correctness of the implementation.

Lemma 26 [Reduction of Ordering over AS_D to Weighted Bipartite Graph Matching] Let β_1 and β_2 be two abstract stores and G be a matching graph of β_1 to β_2. $\beta_1 \sqsubseteq \beta_2$ if and only if G has a matching of weight $|\beta_1| + |\texttt{Def}(\beta_2)|$.
Proof See Appendix A □

Theorem 27 The implementation of ordering satisfies its definition.
Proof This follows from Lemma 26 and the fact that there cannot exist a matching of cost greater than $|\beta_1| + |\texttt{Def}(\beta_2)|$, when testing $\beta_1 \sqsubseteq \beta_2$. □

The following result gives the complexity of the ordering implementation.

Theorem 28 Let β_1 and β_2 be two abstract stores. The complexity of checking if $\beta_1 \sqsubseteq \beta_2$ is not more than $O(|\beta_1|^2|\beta_2|^2 \log |\beta_1||\beta_2|)$.
Proof This follows from [8] which proved that the weighted bipartite matching problem can be solved in time $O(|V|^2 \log |V| + |V||E|)$ i.e $O(|\beta_1|^2|\beta_2|^2 \log |\beta_1||\beta_2|)$ and the definition of the matching graph. □

4.2 Addition

We now turn to the basic operation of CLP languages: adding a constraint to the store. We make the operation slightly more general than needed to simplify the rest of the paper. The operation is specified as follows.

Specification 29 [Adding An Abstract Constraint With Multiplicity] $\uplus : AS_D \times AM_D \rightarrow AS_D$ should satisfy the following consistency condition.

$$\forall \theta_1, \theta_2 \in CS_D, \forall \beta \in AS_D, \forall \gamma \in AM_D : \theta_1 \in Cc(\beta) \ \wedge \ \theta_2 \in Cc(\gamma) \Rightarrow \theta_1 \mathbin{\underline{\cup}} \theta_2 \ \in \ Cc(\beta \uplus \gamma).$$

Definition 30 Let β be an abstract store and γ be an abstract constraint with multiplicity. The operation to add an abstract constraint to an abstract store is defined by

$$\beta \uplus \gamma = \begin{cases} \beta \cup \{\gamma\} & \text{if } \gamma \notin \beta \\ \beta \cup \{\langle \sigma, Any \rangle\} & \text{if } \gamma = \langle \sigma, \mu \rangle \in \beta. \end{cases}$$

Informally speaking, the operation adds γ to β if γ is not in β. Otherwise, the multiplicity needs to be adjusted to take into account the new constraint. This is done by setting the multiplicity to *Any*. Although the operation is very simple, the proof is non-trivial and indicates why it is convenient to consider multisets (and not sets) of constraints in the concrete semantics.[4]

Theorem 31 Let β be an abstract store and γ be an abstract constraint with multiplicity. If $\theta_1 \in Cc(\beta)$ and $\theta_2 \in Cc(\gamma)$, then $\theta_1 \mathbin{\underline{\cup}} \theta_2 \in Cc(\beta \uplus \gamma)$.
Proof See Appendix A. □

Operation $\uplus$ is useful for the implementation of other operations. In fact, it is convenient to generalize it further.

Definition 32 Let β and β' be abstract stores.

$$\beta \biguplus \beta' = \begin{cases} \beta & \text{if } \beta' = \emptyset \\ (\beta \uplus \gamma) \biguplus \beta'' & \text{if } \beta' = \{\gamma\} \cup \beta'', \gamma \notin \beta''. \end{cases}$$

Lemma 33 Let β and β' be abstract stores. If $\theta \in Cc(\beta)$ and $\theta' \in Cc(\beta')$ then $\theta \mathbin{\underline{\cup}} \theta' \in Cc(\beta \biguplus \beta')$.

4.3 Upper Bound

We now turn to the upper bound operation UNION: $AS_D \times AS_D \rightarrow AS_D$.

Specification 34 UNION: $AS_D \times AS_D \rightarrow AS_D$ should satisfy the following consistency condition.

$$\forall \theta \in CS_D, \forall \beta_1 \in AS_D, \forall \beta_2 \in AS_D : \theta \in Cc(\beta_1) \vee \theta \in Cc(\beta_2) \Rightarrow \theta \in Cc(\texttt{UNION}(\beta_1, \beta_2)).$$

We first define UNION of signs and multiplicities.

[4]It is in fact possible to be more precise in certain cases by using multiplicity *ZeroOrOne* when $\langle \sigma, ZeroOrOne \rangle$ and $\langle \sigma, Any \rangle$ are not in β. We kept the above definition for simplicity.

Definition 35 [Upper Bound on Signs] The upper bound operation on signs $\sqcup$: `Sign` $\times$ `Sign` $\rightarrow$ `Sign` is defined as

$$s_1 \sqcup s_2 = \begin{cases} s_1 & \text{if } s_1 = s_2 \\ \top & \text{otherwise.} \end{cases}$$

Definition 36 [Upper Bound on Multiplicities] The upper bound operation on multiplicities $\sqcup$: `Mult` $\times$ `Mult` $\rightarrow$ `Mult` is defined as $\mu_1 \sqcup \mu_2 = max(\mu_1, \mu_2)$.

We now turn to the upper bound operation on abstract stores. Note first that `LSign` has no least upper bound operation. If β_1 is $\{\langle \oplus = \oplus x_1 + \oplus x_2, One\rangle\}$ and β_2 is $\{\langle \oplus = \oplus x_1 + \ominus x_2, One\rangle\}$, both

$$\beta_3 = \{\langle \oplus = \oplus x_1 + \oplus x_2, ZeroOrOne\rangle, \langle \oplus = \oplus x_1 + \ominus x_2, ZeroOrOne\rangle\}$$

and

$$\beta_4 = \{\langle \oplus = \oplus x_1 + \top x_2, One\rangle\}$$

are upper bounds of β_1 and β_2 but they are incomparable. This problem cannot be avoided and indicates the inherent tradeoff between the accuracy of the signs and the accuracy of the multiplicity. In practice, one should choose an ordering appropriate to the application at hand. For this reason, we design a general scheme to generate upper bound operations. Any operation built along the scheme is an upper bound.

Implementation 37 [Upper Bound on Abstract Stores] An upper bound operation on abstract stores `UNION`: $AS_D \times AS_D \rightarrow AS_D$ is any operation obtained by applying in a nondeterministic way the following set of rules.

1 $\quad \mathtt{UNION}(\emptyset, \emptyset) = \emptyset$

2 i $\quad \mathtt{UNION}(\{\langle\sigma, One\rangle\} \cup \beta_1', \beta_2) = \mathtt{UNION}(\beta_1', \beta_2) \uplus \{\langle \sigma, ZeroOrOne\rangle\} \quad \text{if } \langle \sigma, One\rangle \notin \beta_1'$

ii $\quad \mathtt{UNION}(\beta_1, \{\langle\sigma, One\rangle\} \cup \beta_2') = \mathtt{UNION}(\beta_1, \beta_2') \uplus \{\langle \sigma, ZeroOrOne\rangle\} \quad \text{if } \langle \sigma, One\rangle \notin \beta_2'$

3 i $\quad \mathtt{UNION}(\{\langle\sigma, \mu\rangle\} \cup \beta_1', \beta_2) = \mathtt{UNION}(\beta_1', \beta_2) \uplus \{\langle \sigma, \mu\rangle\} \quad \text{if } \langle \sigma, \mu\rangle \notin \beta_1' \wedge \mu \neq One$

ii $\quad \mathtt{UNION}(\beta_1, \{\langle\sigma, \mu\rangle\} \cup \beta_2') = \mathtt{UNION}(\beta_1, \beta_2') \uplus \{\langle \sigma, \mu\rangle\} \quad \text{if } \langle \sigma, \mu\rangle \notin \beta_2' \wedge \mu \neq One$

4 $\quad \mathtt{UNION}(\{\langle s_0 \; op \; \sum_{i=1}^{n} s_i x_i, \mu\rangle\} \cup \beta_1', \{\langle s_0' \; op \; \sum_{i=1}^{n} s_i' x_i, \mu'\rangle\} \cup \beta_2') =$

$$\mathtt{UNION}(\beta_1', \beta_2') \uplus \{\langle (s_0 \sqcup s_0') \; op \; \sum_{i=1}^{n} (s_i \sqcup s_i') x_i, \mu \sqcup \mu'\rangle\}$$

$$\text{if } \langle s_0 \; op \; \sum_{i=1}^{n} s_i x_i, \mu\rangle \notin \beta_1' \text{ and } \langle s_0' \; op \; \sum_{i=1}^{n} s_i' x_i, \mu'\rangle \notin \beta_2'$$

Rules $2i$ and $2ii$ trade the precision of the multiplicity information for the precision of the coefficients, since the constraint is no longer required but its coefficients remain the same. Rules $3i$ and $3ii$ do not lose information. Rule 4 trades the precision of the coefficients for the precision of the multiplicity. It is appropriate whenever μ and μ' are not Any, since they preserve the information that at most one constraint (or possibly exactly one) is represented.

Theorem 38 Let β_1 and β_2 be abstract stores and θ be a constraint store.

$$\theta \in Cc(\beta_1) \vee \theta \in Cc(\beta_2) \;\Rightarrow\; \theta \in Cc(\mathtt{UNION}(\beta_1, \beta_2)).$$

Proof See Appendix A. □

4.4 Projection

We now turn to the projection operation. Figure 1 describes a simple projection algorithm based on the traditional Gaussian and Fourier eliminations which are standard in this area. Figure 2 presents the abstract algorithm. The algorithms are close to those in [18] but they are simplified thanks to the introduction of operations `Csplit` and `Asplit` which avoids much of the tedious case analysis. Fourier elimination is also more precise.

The intuition behind the concrete version is as follows. `Cproject` nondeterministically chooses a constraint in the store. If the constraint is an equation whose coefficient for x_v is non-zero, Gaussian elimination is performed. Otherwise, Fourier elimination is used. Gaussian elimination starts by partitioning the store in three sets θ^0, θ^+ and θ^- which respectively contain constraints whose coefficient for x_v is 0, strictly positive, or strictly negative. It then eliminates x_v by using the constraint or its negation on θ^+ and θ^- using operation `Ccombine` on a pair of constraints. Fourier elimination considers each constraint in turn, partitions the store once again, and uses Fourier elimination on the compatible pairs. The presentation is slightly more complicated than required in order to make the abstract version simpler.

The abstract version mimics almost line by line the concrete version showing the benefits of using operations `Csplit` and `Asplit`. The only place where we deal with $\top$ is precisely in `Asplit`. Of course, in the abstract algorithm, operations and relations on signs replace operations and relations on coefficients. The correctness of the abstract version can be proven by showing the consistency of the various operations. Operation `Acombine` mimics `Ccombine` by performing operations on signs instead of on coefficients. It satisfies the following consistency condition.

Lemma 39 [Combine] Let σ_1, σ_2 be abstract constraints, λ_1, λ_2 be constraints and $v \in \mathcal{N}$.

$$\forall \lambda_1 \in Cc(\sigma_1), \lambda_2 \in Cc(\sigma_2) : \mathtt{Ccombine}(\lambda_1, \lambda_2, v) \in Cc(\mathtt{Acombine}(\sigma_1, \sigma_2, v)).$$

Proof Direct consequence of the consistency of sign arithmetic. □

Operation `Csplit` partitions the store into three sets depending upon the coefficient of c_v. Its abstract version needs to deal with the case where the coefficient of the abstract constraint is $\top$. In this case, specializations of the abstract constraint are inserted in all three sets and its multiplicity is adjusted to reflect that the information is not sure. Its consistency condition is as follows.

Lemma 40 [Split] Let θ be a constraint store, let β be an abstract store, and $v \in \mathcal{N}$. We have

$$\left.\begin{array}{l} \mathtt{Csplit}(\theta, v) = \langle \theta^0, \theta^+, \theta^- \rangle \\ \theta \in Cc(\beta) \\ \mathtt{Asplit}(\beta, v) = \langle \beta^0, \beta^+, \beta^- \rangle \end{array}\right\} \Rightarrow \theta^0 \in Cc(\beta^0) \;\wedge\; \theta^+ \in Cc(\beta^+) \;\wedge\; \theta^- \in Cc(\beta^-).$$

Proof See Appendix A. □

Cproject(θ,v) {
let $\theta \ = \ \theta' \ \underline{\cup} \ \{\lambda\}$ **in**
    case op(λ) is '=' $\wedge \ \lambda[v] > 0$:
        return Cgauss($\theta \setminus \{\lambda\}$,$\lambda$,v);
    case op(λ) is '=' $\wedge \ \lambda[v] < 0$:
        return Cgauss($\theta \setminus \{\lambda\}$,Cnegate($\lambda$),v);
    otherwise:
        return Cfourier(θ,v);
}

Cgauss(θ,λ,v) {
$\langle \theta^0, \theta^+, \theta^- \rangle$:= Csplit(θ,v);
return $\theta^0 \ \underline{\cup}$ Cgauss_step(θ^+,Cnegate(λ),v)
            $\underline{\cup}$ Cgauss_step(θ^-,λ,v);
}

Cgauss_step(θ,λ,v) {
if $\theta = \emptyset$ **return** $\emptyset$;
else let $\theta = \{\lambda'\} \ \underline{\cup} \ \theta'$ **in**
    return Cgauss_step(θ',λ,v) $\underline{\cup}$
            {Ccombine(λ',λ,v)};
}

Cfourier(θ,v) {
if $\theta = \emptyset$ **then return** $\emptyset$
else let $\theta = \{\sigma'\} \underline{\cup} \theta'$ **in**
    $\langle \theta_1^0, \theta_1^+, \theta_1^- \rangle$:= Csplit($\{\sigma'\}, v$);
    $\langle \theta_2^0, \theta_2^+, \theta_2^- \rangle$:= Csplit(θ',v);
    return $\theta_1^0 \ \underline{\cup}$ Cfourier_step(θ_1^+,θ_2^-,v)
                $\underline{\cup}$ Cfourier_step(θ_2^+,θ_1^-,v)
                $\underline{\cup}$ Cfourier(θ',v);
}

Cfourier_step(θ^+,θ^-,v) {
$\theta := \emptyset$;
for each $\lambda^+ \in \theta^+$ **and** $\lambda^- \in \theta^-$
    $\theta := \theta \ \underline{\cup}$ {Ccombine(λ^-,λ^+,v)};
return θ;
}

Ccombine(λ_1,λ_2,v) {
for $i := 0, \ldots, n$
    λ[i] := $(-\lambda_2[v]) \times \lambda_1[i] + \lambda_1[v] \times \lambda_2[i]$;
λ[v] := 0;
op(λ) := op(λ_1) $\bowtie$ op(λ_2);
return λ;
}

Cnegate(λ) {
for $i = 0, \ldots, n$
    λ'[i] := - λ[i];
op(λ') := op(λ);
return λ';
}

$\bowtie$	$<$	$\leq$	$=$
$<$	$<$	$<$	$<$
$\leq$	$<$	$\leq$	$\leq$
$=$	$<$	$\leq$	$=$

Csplit(θ,v) {
if $\theta = \emptyset$ **return** $\langle \emptyset, \emptyset, \emptyset \rangle$;
else let $\theta = \{\lambda\} \ \underline{\cup} \ \theta'$
            $\langle \theta^0, \theta^+, \theta^- \rangle$ = Csplit(θ',v) **in**
    case $\lambda[v] = 0$: **return** $\langle \theta^0 \ \underline{\cup} \ \{\lambda\}, \theta^+, \theta^- \rangle$;
    case $\lambda[v] > 0$: **return** $\langle \theta^0, \theta^+ \ \underline{\cup} \ \{\lambda\}, \theta^- \rangle$;
    case $\lambda[v] < 0$: **return** $\langle \theta^0, \theta^+, \theta^- \ \underline{\cup} \ \{\lambda\} \rangle$;
}

Figure 1: Concrete Projection Algorithms

Aproject(β,v) {
let $\beta = \{\gamma\} \cup \beta' \ \wedge\ \gamma \notin \beta \ \wedge\ \gamma = \langle \sigma, \mu \rangle$ **in**
  case $\mu = \mathit{One} \wedge \mathrm{op}(\sigma)$ is '=' $\wedge\ \sigma[v] = \oplus$:
    return Agauss($\beta \setminus \{\gamma\}$,$\sigma$,v);
  case $\mu = \mathit{One} \wedge \mathrm{op}(\sigma)$ is '=' $\wedge\ \sigma[v] = \ominus$:
    return Agauss($\beta \setminus \{\gamma\}$,Anegate($\sigma$),v);
  otherwise:
    return Afourier(β,v);
}

Agauss(β,σ,v) {
$\langle \beta^0, \beta^+, \beta^- \rangle$:= Asplit(β,v);
return $\beta^0 \uplus$ Agauss_step(β^+,Anegate(σ),v)
    $\uplus$ Agauss_step(β^-,σ,v);
}

Agauss_step(β,σ,v) {
if $\beta = \emptyset$ **return** $\emptyset$;
else let $\beta = \{\gamma'\} \cup \beta', \gamma' = \langle \sigma', \mu' \rangle \notin \beta'$ **in**
  return Agauss_step(β',σ,v) $\uplus$
    $\{\langle$ Acombine(σ',σ,v), $\mu' \rangle\}$;
}

Afourier(β,v) {
if $\beta = \emptyset$ **then return** $\emptyset$
else let $\beta = \{\gamma'\} \cup \beta', \gamma' = \langle \sigma', \mu' \rangle \notin \beta'$ **in**
  $\langle \beta_1^0, \beta_1^+, \beta_1^- \rangle$:= Asplit($\{\gamma'\}, v$);
  $\langle \beta_2^0, \beta_2^+, \beta_2^- \rangle$:= Asplit(β',v);
  if $\mu' = \mathit{Any}$ **then return**
    $\beta_1^0 \uplus$ Afourier_step(β_1^+,$\beta_2^- \uplus \beta_1^-$,v)
      $\uplus$ Afourier_step($\beta_2^+ \uplus \beta_1^+$,$\beta_1^-$,v)
      $\uplus$ Afourier(β',v)
  else return
    $\beta_1^0 \uplus$ Afourier_step(β_1^+,β_2^-,v)
      $\uplus$ Afourier_step(β_2^+,β_1^-,v)
      $\uplus$ Afourier(β',v)
}

Afourier_step(β^+,β^-,v) {
$\beta := \emptyset$;
for each $\langle \sigma^+, \mu^+ \rangle \in \beta^+$ **and** $\langle \sigma^-, \mu^- \rangle \in \beta^-$
  $\beta := \beta \uplus \{\langle$ Acombine(σ^+, σ^-,v), $\mu^+ \sqcup \mu^- \rangle\}$;
return β;
}

Acombine(σ_1,σ_2,v) {
for $i := 0, \ldots, n$
  σ[i] := $(-\sigma_2[v]) \times \sigma_1[i] + \sigma_1[v] \times \sigma_2[i]$;
σ[v] := 0;
op(σ) := op(σ_1) $\bowtie$ op(σ_2);
return σ;
}

Anegate(σ) {
for $i = 0, \ldots, n$
  σ'[i] := - σ[i];
op(σ') := op(σ);
return σ';
}

Asplit(β,v) {
if $\beta = \emptyset$ **return** $\langle \emptyset, \emptyset, \emptyset \rangle$;
else let $\beta = \{\gamma\} \cup \beta' \ \wedge\ \gamma \notin \beta' \ \wedge\ \gamma = \langle \sigma, \mu \rangle$
    $\gamma^0 = \langle \sigma^0, \mu \sqcup \mathit{ZeroOrOne} \rangle$
    $\gamma^+ = \langle \sigma^+, \mu \sqcup \mathit{ZeroOrOne} \rangle$
    $\gamma^- = \langle \sigma^-, \mu \sqcup \mathit{ZeroOrOne} \rangle$
    $\sigma^0 = \sigma$ except that $\sigma^0[v] = 0$
    $\sigma^+ = \sigma$ except that $\sigma^+[v] = \oplus$
    $\sigma^- = \sigma$ except that $\sigma^-[v] = \ominus$
    $\langle \beta^0, \beta^+, \beta^- \rangle$ = Asplit(β',v) **in**
  case $\sigma[v] = 0$:
    return $\langle \beta^0 \uplus \gamma, \beta^+, \beta^- \rangle$;
  case $\sigma[v] = \oplus$:
    return $\langle \beta^0, \beta^+ \uplus \gamma, \beta^- \rangle$;
  case $\sigma[v] = \ominus$:
    return $\langle \beta^0, \beta^+, \beta^- \uplus \gamma \rangle$;
  case $\sigma[v] = \top$:
    return $\langle \beta^0 \uplus \gamma^0, \beta^+ \uplus \gamma^+, \beta^- \uplus \gamma^- \rangle$;
}

Figure 2: Abstract Projection Algorithms

Operation `Agauss` mimics operation `Cgauss` line by line and its consistency condition is as follows.

Lemma 41 [Gauss] Let $v \in \mathcal{N}$, σ be an abstract constraint such that $op(\sigma)$ is '=' and $\sigma[v] = \oplus$, and β be an abstract store. We have

$$\theta \in Cc(\beta) \wedge \lambda \in Cc(\sigma) \Rightarrow \texttt{Cgauss}(\theta, \lambda, v) \in Cc(\texttt{Agauss}(\beta, \sigma, v)).$$

Proof See Appendix A. □

Similarly, operation `Afourier` closely mimics operation `Cfourier`. The main difference comes from the fact that we avoid combining a constraint with multiplicity *One* or *ZeroOrOne* with itself, contrary to the algorithm in [18]. This is achieved by testing the multiplicity of the constraint.

Lemma 42 [Fourier] Let $v \in \mathcal{N}$, θ be a store, and β be an abstract store. We have

$$\theta \in Cc(\beta) \Rightarrow \texttt{Cfourier}(\theta, v) \in Cc(\texttt{Afourier}(\beta, v)).$$

Proof See Appendix A. □

We now prove the main result of this section: the consistency of projection. It is sufficient to prove that there exists a projection algorithm whose result is approximated by `Aproject`, since the projection algorithm can be arbitrary (and in fact is only used in an actual implementation to present the results).

Theorem 43 [Project] Let $v \in \mathcal{N}$, θ be a store and β be an abstract store.

$$\theta \in Cc(\beta) \;\Rightarrow\; \text{there exists } \theta' \text{ such that } \theta' \leftrightarrow \texttt{Cproject}(\theta, \texttt{v}) \;\wedge\; \theta' \in Cc(\texttt{Aproject}(\beta, v)).$$

Proof See Appendix A. □

5 Conclusion

This paper reconsidered the `LSign` domain of [18] and was motivated by the fact that the ordering relation given in [18] does not capture the intended meaning. Our first contribution was to define a new ordering on `LSign`, whose meaning is now defined by a monotone concretization function from abstract constraint stores to sets of constraint stores (which are multisets of constraints). The ordering and the concretization function capture the intended meaning of [18]. Our second contribution is to show that ordering in `LSign` reduces to a polynomial matching problem. Our third contribution is to define implementations of the abstract operations, i.e. abstract addition of a constraint, abstract projection and a schema to generate upper bound operations, and to prove their consistency.

Acknowledgments

We would like to thank K. Marriott for responding quickly to our query about the correctness of the ordering in `LSign` and B. Le Charlier for being a constant source of inspiration and for his numerous helpful comments on a draft of this paper. This research was supported in part by the Office of Naval Research under grant ONR Grant N00014-94-1-1153, the National Science Foundation under grant numbers CCR-9357704, a NSF National Young Investigator Award with matching funds of Hewlett-Packard.

References

[1] C. Braem, B. Le Charlier, S. Modart, and P. Van Hentenryck. Cardinality Analysis of Prolog. In *Proceedings of the International Symposium on Logic Programming (ILPS-94)*, pages 457–471, Ithaca, NY, November 1994.

[2] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10(2):91–124, February 1991.

[3] W. Buttner and H. Simonis. Embedding Boolean Expressions into Logic Programming. *Journal of Symbolic Computation*, 4:191–205, October 1987.

[4] A. Colmerauer. An Introduction to Prolog III. *CACM*, 28(4):412–418, 1990.

[5] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In New York ACM Press, editor, *Conf. Record of Fourth ACM Symposium on Programming Languages (POPL'77)*, pages 238–252, Los Angeles, CA, January 1977.

[6] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Conf. Record of Fifth ACM Symposium on Principles of Programming Languages (POPL'78)*, 1978.

[7] V. Dumortier and G. Janssens. Towards a Practical Full Mode Inference System for CLP(H,N). In *Eleventh International Conference on Logic Programming (ICLP-94)*, Santa Marguerita Ligure, Italy, June 1994.

[8] M.L. Fredman and R.E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *JACM*, 34:596–615, July 1987.

[9] M. Garcia de la Banda and M. Hermenegildo. A Practical Approach to the Global Analysis of CLP Programs. In *Proc. of the International Symposium on Logic Programming (ILPS'93)*, Vancouver, Canada, November 1993.

[10] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *POPL-87*, Munich, FRG, January 1987.

[11] J. Jaffar, S. Michaylov, P.J. Stuckey, and R. Yap. The CLP($\Re$) Language and System. *ACM Transactions on Programming Languages*, 14(3):339–395, 1992.

[12] B. Le Charlier, S. Rossi, and P. Van Hentenryck. An Abstract Interpretation Framework Which Accurately Handles Prolog Search Rule and the Cut. In *Proceedings of the International Symposium on Logic Programming (ILPS-94)*, pages 157–171, Ithaca, NY, November 1994.

[13] B. Le Charlier and P. Van Hentenryck. A Universal Top-Down Fixpoint Algorithm. Technical Report CS-92-25, CS Department, Brown University, 1992.

[14] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, January 94.

[15] K. Marriott and H. Sondergaard. Notes for a Tutorial on Abstract Interpretation of Logic Programs. North American Conference on Logic Programming, Cleveland, Ohio, October 1989.

[16] K. Marriott and H. Sondergaard. Analysis of Constraint Logic Programs. In *Proceedings of the North American Conference on Logic Programming (NACLP-90)*, Austin, TX, October 1990.

[17] K. Marriott and P. Stuckey. The 3 R's of optimizing Constraint Logic Programs: Refinement, Removal, and Reordering. In *Proc. of the 20th ACM Symposium on Principles of Programming Languages (POPL'93)*, Charleston, South Carolina, January 1993.

[18] K. Marriott and P. Stuckey. Approximating Interaction Between Linear Arithmetic Constraints. In *Proc. of the International Symposium on Logic Programming (ILPS'94)*, Ithaca, NY, November 1994.

[19] K. Muthukumar and M. Hermenegildo. Compile-Time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2-3):315–347, August 1992.

[20] W. Older and A. Vellino. Extending Prolog with Constraint Arithmetics on Real Intervals. In *Canadian Conference on Computer & Electrical Engineering*, Ottawa, 1990.

[21] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series, The MIT Press, Cambridge, MA, 1989.

[22] P. Van Hentenryck and T. Graf. Standard Forms for Rational Linear Arithmetics in Constraint Logic Programming. *Annals of Mathematics and Artificial Intelligence*, 5(2-4):303–320, 1992.

[23] W. Winsborough. Multiple Specialization using Minimal-Function Graph Semantics. *Journal of Logic Programming*, 13(4), July 1992.

A Proofs of the Main Results

Lemma 44 [Composition of Ordering Functions] Let $\beta_1, \beta_2, \beta_3$ be abstract stores, let f be an ordering function of β_1 to β_2, and let g be an ordering function of β_2 to β_3. The composition $g \circ f$ is an ordering function of β_1 to β_3.
Proof The three properties of ordering functions hold.

1. $\forall \gamma \in \beta_1 : \gamma \sqsubseteq f(\gamma) \sqsubseteq g(f(\gamma))$.
2. By hypothesis, we have $\forall \gamma_1, \gamma_2 \in \beta_1 : \gamma_1 \neq \gamma_2 \Rightarrow f(\gamma_1) \neq f(\gamma_2) \vee f(\gamma_1) \in \mathtt{Indef}(\beta_2)$ and $\forall \gamma_1, \gamma_2 \in \beta_2 : \gamma_1 \neq \gamma_2 \Rightarrow g(\gamma_1) \neq g(\gamma_2) \vee g(\gamma_1) \in \mathtt{Indef}(\beta_3)$. By combination of the two statements and $f(\gamma_1) \sqsubseteq (g \circ f)(\gamma_1)$, we obtain $\forall \gamma_1, \gamma_2 \in \beta_1 : \gamma_1 \neq \gamma_2 \Rightarrow (g \circ f)(\gamma_1) \neq (g \circ f)(\gamma_2) \vee (g \circ f)(\gamma_1) \in \mathtt{Indef}(\beta_3)$.
3. By hypothesis, we have $\mathtt{Def}(\beta_3) \subseteq \mathit{codomain}(g)$. By property 1 of Definition 13, there exist $\gamma_1, \ldots, \gamma_n$ in $\mathtt{Def}(\beta_2)$ such that $\{f(\gamma_1), \ldots, f(\gamma_n)\} = \mathtt{Def}(\beta_3)$. The result follows from the hypothesis that $\mathtt{Def}(\beta_2) \subseteq \mathit{codomain}(f)$.

□

Theorem 16 $\sqsubseteq$: $AS_D \times AS_D$ is a partial order.
Proof Reflexivity follows by choosing the identity function as ordering function. Transitivity follows by Lemma 44. To show anti-symmetry, i.e.

$$\beta_1 \subseteq \beta_2 \ \wedge \ \beta_2 \subseteq \beta_1 \Rightarrow \beta_1 = \beta_2,$$

let $f : \beta_1 \to \beta_2$ and $g : \beta_2 \to \beta_1$ be ordering functions of β_1 to β_2 and of β_2 to β_1 respectively. By Lemma 44, $h = g \circ f : \beta_1 \to \beta_1$ is an ordering function of β_1 to β_1.

Consider now an element $\gamma = \langle \sigma, \mathit{Any} \rangle$ in β_1. $h(\gamma) = \gamma$, since otherwise we would have $\gamma \sqsubseteq h(\gamma)$ and $h(\gamma) \neq \gamma$, which contradicts the definition of abstract store.

Consider now an element $\gamma = \langle \sigma, \mathit{One} \rangle$ in β_1. We prove that $h(\gamma) = \gamma$. First note that, for all $\gamma \in \mathtt{Def}(\beta_1)$, $h(\gamma)$ must be of the form $\langle \sigma', \mathit{One} \rangle$, since h is injective and only elements of $\mathtt{Def}(\beta_1)$ can map into elements of $\mathtt{Def}(\beta_1)$ by Definition 13. Hence, when restricted to $\mathtt{Def}(\beta_1)$, h is a bijection. Assume now that there exists γ in $Def(\beta_1)$ such that $h(\gamma) \neq \gamma$. We can thus construct a sequence

$$\begin{array}{lcl} \gamma_0 & = & \gamma \\ \gamma_i & = & h(\gamma_{i-1}) \quad i > 0. \end{array}$$

Since h is a bijection, there exists a $k > 1$ such that $\gamma_k = \gamma_0$. By Definition 13, we have that

$$\gamma_0 \sqsubseteq \gamma_1 \sqsubseteq \ldots \sqsubseteq \gamma_0$$

which implies that $\gamma_0 = \gamma_1$ contradicting our assumption.

Consider now an element $\gamma = \langle \sigma, \mathit{ZeroOrOne} \rangle$ in β_1. We prove that $h(\gamma) = \gamma$. Since h is a bijection when restricted to $\mathtt{Def}(\beta_1)$, only elements $\mathtt{Pos}(\beta_1)$ can map into elements of $\mathtt{Pos}(\beta_1)$ by Definition 13. Moreover, elements of $\mathtt{Pos}(\beta_1)$ cannot map into elements of $\mathtt{Indef}(\beta_1)$ by the definition of abstract stores (subsumption condition). h is thus a bijection on $\mathtt{Pos}(\beta_1)$ by Definition 13 since h is injective on $\mathtt{Pos}(\beta_1)$. Hence we can apply the same reasoning as for the definite case. □

Lemma 19 Let β be an abstract store and θ a concrete constraint store. $\theta \in Cc(\beta)$ if and only if there exists a lifting function f of θ to β.
Proof By induction on $|\beta|$.

Basis: $\beta = \emptyset$. If $\theta \in Cc(\beta)$ then $\theta = \emptyset$ by Definition 10 and we choose the empty function as a lifting function. Conversely, if there exists a function $f : \theta \to \beta$, then $\theta = \emptyset$ since $\beta = \emptyset$ and f must be the empty function. The result follows from Definition 10.
Induction Step: Assume that the hypothesis holds for all abstract stores whose cardinality is not greater than n. We show that it holds for abstract stores of cardinality $n+1$.

Consider an abstract store β satisfying $|\beta| = n+1$ and let β be $\{\gamma\} \cup \beta'$, where $\gamma \notin \beta'$ and $|\beta'| = n$. By Definition 10,

$$Cc(\beta) = Cc(\{\gamma\} \cup \beta') = \{\ \theta_1 \uplus \theta' \mid \theta_1 \in Cc(\gamma) \wedge \theta' \in Cc(\beta')\ \}.$$

($\Rightarrow$) Let $\theta \in Cc(\beta)$. Hence, $\theta = \theta_1 \uplus \theta'$, where $\theta_1 \in Cc(\gamma) \wedge \theta' \in Cc(\beta')$. By hypothesis, there exists a lifting function f' of θ' to β'. Define a function $f : \theta \to \beta$ as

$$f(\lambda) = \begin{cases} f'(\lambda) & \text{if } \lambda \in \theta' \\ \gamma & \text{if } \lambda \in \theta_1. \end{cases}$$

We show that f is a lifting function of θ to β.

1. $\forall \lambda \in \theta' : \lambda \in Cc(cons(f'(\lambda)))$ by Definition 18
$\forall \lambda \in \theta_1 : \lambda \in Cc(cons(\gamma))$ by Lemma 17
$\Rightarrow \forall \lambda \in \theta : \lambda \in Cc(cons(f(\lambda)))$.

2. $\forall \lambda_1, \lambda_2 \in \theta' : \lambda_1 \neq \lambda_2 \Rightarrow f'(\lambda_1) \neq f'(\lambda_2) \vee f'(\lambda_1) \in \mathtt{Indef}(\beta')$ by Definition 18
$\Rightarrow \forall \lambda_1, \lambda_2 \in \theta' : \lambda_1 \neq \lambda_2 \Rightarrow f(\lambda_1) \neq f(\lambda_2) \vee f(\lambda_1) \in \mathtt{Indef}(\beta)$ since $\mathtt{Indef}(\beta') \subseteq \mathtt{Indef}(\beta)$

$\forall \lambda_1 \in \theta_1, \lambda_2 \in \theta', \lambda_1 \neq \lambda_2 \Rightarrow f(\lambda_1) \neq f(\lambda_2)$ since $f(\lambda_1) = \gamma \notin \beta'$ and $f(\lambda_2) \in \beta'$

$\forall \lambda_1, \lambda_2 \in \theta_1 : \lambda_1 \neq \lambda_2 \Rightarrow \gamma$ is indefinite by Definition 7
$\Rightarrow \forall \lambda_1, \lambda_2 \in \theta_1 : \lambda_1 \neq \lambda_2 \Rightarrow f(\lambda_1) \in \mathtt{Indef}(\beta)$ since $\gamma = f(\lambda_1)$

3. $\mathtt{Def}(\beta') \subseteq codomain(f')$ by Definition 18
γ is not definite $\Rightarrow \mathtt{Def}(\beta) = \mathtt{Def}(\beta') \wedge codomain(f') \subseteq codomain(f)$
$\Rightarrow \mathtt{Def}(\beta) \subseteq codomain(f)$
γ is definite $\Rightarrow \mathtt{Def}(\beta) = \{\gamma\} \cup \mathtt{Def}(\beta') \wedge \{\gamma\} \cup codomain(f') = codomain(f)$
$\Rightarrow \mathtt{Def}(\beta) \subseteq codomain(f)$.

($\Leftarrow$) Let f be an lifting function of θ to $\beta = \{\gamma\} \cup \beta'$. We show that $\theta \in Cc(\beta)$.

Case γ is not definite. Let $\lambda_1, \ldots, \lambda_m$ be all elements in θ such that $f(\lambda_i) = \gamma$ $(1 \leq i \leq m)$. $\lambda_i \in Cc(cons(\gamma))$ $(1 \leq i \leq m)$ by Definition 18 and $\{\lambda_1, \ldots, \lambda_m\} \in Cc(\gamma)$ by Definition 7. Let $\theta = \{\lambda_1, \ldots, \lambda_m\} \uplus \theta'$ and consider the function $f' : \theta' \to \beta'$ defined by $f'(\lambda) = f(\lambda)$ for all $\lambda \in \theta'$. We show that f' is a lifting function of θ' to β'.

1. $\forall \lambda \in \theta'$: $\lambda \in Cc(cons(f'(\lambda)))$ by Definition 18.

2. $\forall \lambda_1, \lambda_2 \in \theta' : \lambda_1 \neq \lambda_2 \Rightarrow f(\lambda_1) \neq f(\lambda_2) \vee f(\lambda_1) \in \texttt{Indef}(\beta)$ by Definition 18
$\forall \lambda_1, \lambda_2 \in \theta' : \lambda_1 \neq \lambda_2 \Rightarrow f'(\lambda_1) \neq f'(\lambda_2) \vee f'(\lambda_1) \in \texttt{Indef}(\beta')$
since $\texttt{Indef}(\beta') = \texttt{Indef}(\beta) \setminus \{\gamma\}$ and $f(\lambda_1) \neq \gamma$.

3. $\texttt{Def}(\beta) \subseteq codomain(f)$ by Definition 18
$codomain(f') = codomain(f) \setminus \{\gamma\}$
$\texttt{Def}(\beta') = \texttt{Def}(\beta) \Rightarrow \texttt{Def}(\beta') \subseteq codomain(f)$ since γ is not definite
$\Rightarrow \texttt{Def}(\beta') \subseteq codomain(f')$ since $\gamma \notin \texttt{Def}(\beta')$.

By hypothesis, $\theta' \in Cc(\beta')$. By Definition 10, $\{\lambda_1, \ldots, \lambda_m\} \underline{\cup}\, \theta' \in Cc(\{\gamma\} \cup \beta')$.

Case γ is definite. By Definition 18, there exists a unique $\lambda_1 \in \theta$, such that $f(\lambda_1) = \gamma$. $\lambda_1 \in Cc(cons(\gamma))$ by Definition 18 and $\{\lambda_1\} \in Cc(\gamma)$ by Definition 7. Let $\theta = \{\lambda_1\} \underline{\cup}\, \theta'$ and consider the function $f' : \theta' \rightarrow \beta'$ defined by $f'(\lambda) = f(\lambda)$ for all $\lambda \in \theta'$. We show that f' is a lifting function of θ' to β'.

1. $\forall \lambda \in \theta'$, $\lambda \in Cc(cons(f'(\lambda)))$ by Definition 18.

2. $\forall \lambda_1, \lambda_2 \in \theta', \lambda_1 \neq \lambda_2 \Rightarrow f(\lambda_1) \neq f(\lambda_2) \vee f(\lambda_1) \in \texttt{Indef}(\beta)$ by Definition 18
$\forall \lambda_1, \lambda_2 \in \theta', \lambda_1 \neq \lambda_2 \Rightarrow f'(\lambda_1) \neq f'(\lambda_2) \vee f'(\lambda_1) \in \texttt{Indef}(\beta')$ since $\texttt{Indef}(\beta) = \texttt{Indef}(\beta')$.

3. $\texttt{Def}(\beta) \subseteq codomain(f)$ by Definition 18
$codomain(f) = \{\gamma\} \cup codomain(f')$
$\{\gamma\} \cup \texttt{Def}(\beta') = \texttt{Def}(\beta) \Rightarrow \texttt{Def}(\beta') \subseteq codomain(f')$ since γ is definite.

By hypothesis, $\theta' \in Cc(\beta')$. By Definition 10, $\{\lambda_1\} \underline{\cup}\, \theta' \in Cc(\{\gamma\} \cup \beta')$. □

Lemma 26 Let β_1 and β_2 be two abstract stores and G be a matching graph of β_1 to β_2. $\beta_1 \sqsubseteq \beta_2$ if and only if G has a matching of weight $|\beta_1| + |\texttt{Def}(\beta_2)|$.

Proof

($\Rightarrow$) Let $\beta_1 \sqsubseteq \beta_2$. By Definition 14, there exists an order mapping f of β_1 to β_2. Consider the set given by $M = M_2 \cup M_3 \cup M_4$ where,

$$\begin{aligned} M_2 &= \{(\gamma_1, \gamma_2) \mid \gamma_1 \in V_1 \wedge \gamma_2 \in V_2 \wedge \gamma_2 = f(\gamma_1)\} \\ M_3 &= \{(\gamma_1, \gamma_2) \mid \gamma_1 \in V_1 \wedge \gamma_2 \in V_3 \wedge \gamma_2 = f(\gamma_1)\} \\ M_4 &= \{(\gamma_1, \gamma_2^{\gamma_1}) \mid \gamma_1 \in V_1 \wedge {\gamma_2}^{\gamma_1} \in V_4 \wedge \gamma_2 = f(\gamma_1)\} \end{aligned}$$

We first prove that M is a matching. By Definition 13, for each edge in M, we have $\gamma_1 \sqsubseteq f(\gamma_1) = \gamma_2$. Therefore, $M_i \subseteq E_i$ $(2 \leq i \leq 4)$, i.e., M is a set of edges of G. In addition,

1. each vertex γ_1 in V_1 appears in exactly one edge in M, since f is an ordering function (property 1);

2. each vertex γ_2 in V_2 appears in exactly one edge in M_2 by Definition 13 (properties 2 and 3);

3. each vertex γ_2 in V_3 appears in at most one edge in M_3 by Definition 13 (property 2);

4. each vertex ${\gamma_2}^{\gamma_1}$ in V_4 appears in at most one edge in M_4, by definition of M_4 since f is a function.

Hence, M is a matching of G.

We now prove that M has the appropriate weight. By point 1 above, M has exactly $|V_1|$ edges. By point 2, M_2 has exactly $|V_2|$ edges. Therefore M has $|V_2|$ edges in M_2 and $|V_1| - |V_2|$ edges in $M_3 \cup M_4$. Hence M is a matching of weight $2|V_2|+(|V_1|-|V_2|)$, i.e. $|V_1|+|V_2|$, i.e. $|\beta_1|+|\mathtt{Def}(\beta_2)|$.

($\Leftarrow$). Let G have a matching M of weight $|\beta_1| + |\mathtt{Def}(\beta_2)|$. Then $M = M_2 \cup M_3 \cup M_4$ where

$$\begin{array}{rcl} M_2 & = & \{(\gamma_1, \gamma_2) | \gamma_1 \in V_1 \wedge \gamma_2 \in V_2\} \\ M_3 & = & \{(\gamma_1, \gamma_2) | \gamma_1 \in V_1 \wedge \gamma_2 \in V_3\} \\ M_4 & = & \{(\gamma_1, \gamma_2^{\gamma_1}) | \gamma_1 \in V_1 \wedge {\gamma_2}^{\gamma_1} \in V_4\} \end{array}$$

We define f by $f(\gamma_1) = \gamma_2$ if $(\gamma_1, \gamma_2) \in M_2$, or $(\gamma_1, \gamma_2) \in M_3$, or $(\gamma_1, {\gamma_2}^{\gamma_1}) \in M_4$ and $f(\gamma_1) =$ *undefined* otherwise. We first show that f is a total function from β_1 to β_2, i.e., there is no undefined value. It suffices to show that each vertex in V_1 appears in at least one edge of M. If it is not the case, then there will be n vertices not in M and $|V_1| - n$ in M $(n > 0)$. Since each vertex appears at most once in M and only $|V_2|$ of them can be given a weight of 2, the weight of the matching is only $2|V_2| + |V_1| - n - |V_2|$ which is $|V_2| + |V_1| - n$ contradicting our hypothesis. We now show that f is an ordering function of β_1 to β_2.

1. Property 1 follows from the definition of the edges in the matching graph G;

2. Property 2 follows from the definition of a matching;

3. To show Property 3, assume that there exists $\gamma_2 \in \mathtt{Def}(\beta_2)$ such that $\gamma_2 \notin \mathit{codomain}(f)$. This means that there is no $\gamma_1 \in \beta_1$ such that (γ_1, γ_2) is an edge of G. This implies that the weight of the matching is at best $|V_2| + |V_1| - 1$ which contradicts the hypothesis.

□

Theorem 31 Let β be an abstract store and γ be an abstract constraint with multiplicity. If $\theta_1 \in Cc(\beta)$ and $\theta_2 \in Cc(\gamma)$, then $\theta_1 \,\underline{\cup}\, \theta_2 \in Cc(\beta \uplus \gamma)$.

Proof Let $\gamma = \langle \sigma, \mu \rangle$. If $\theta_1 \in Cc(\beta)$, there exists a lifting function f of θ_1 to β by Lemma 19. Consider the function $f' : \theta_1 \,\underline{\cup}\, \theta_2 \to \beta \uplus \gamma$ as

$$\begin{array}{ll} f'(\lambda) = f(\lambda) & \text{for all } \lambda \in \theta_1. \\ f'(\lambda) = \left\{ \begin{array}{ll} \gamma & \text{if } \gamma \notin \beta \\ \langle \sigma, \mathit{Any} \rangle & \text{if } \gamma = \langle \sigma, \mu \rangle \in \beta. \end{array} \right. & \text{for all } \lambda \in \theta_2. \end{array}$$

We show that f' is a lifting function of $\theta_1 \,\underline{\cup}\, \theta_2$ to $\beta \uplus \gamma$.

1. $\forall \lambda_1 \in \theta_1 : \lambda_1 \in Cc(\mathit{cons}(f'(\lambda_1)))$ by Definition 18 and $f'(\lambda_1) = f(\lambda_1)$
$\forall \lambda_2 \in \theta_2 : \lambda_2 \in Cc(\mathit{cons}(f'(\lambda)))$ by Lemma 17, $\gamma \sqsubseteq f'(\lambda_2)$ and Lemma 5
$\Rightarrow \forall \lambda \in \theta_1 \underline{\cup} \theta_2 : \lambda \in Cc(\mathit{cons}(f'(\lambda)))$.

2. $\forall \lambda_1, \lambda_2 \in \theta_1 : \lambda_1 \neq \lambda_2 \Rightarrow$

$$
\begin{array}{ll}
\quad f(\lambda_1) \neq f(\lambda_2) \vee f(\lambda_1) \in \mathtt{Indef}(\beta) & \text{by Definition 18} \\
\quad f'(\lambda_1) \neq f'(\lambda_2) \vee f'(\lambda_1) \in \mathtt{Indef}(\beta \uplus \gamma) & \text{since } f'(\lambda_1) = f(\lambda_1), f'(\lambda_2) = f(\lambda_2), \beta \subseteq \beta \uplus \gamma \\
\gamma \notin \beta \;\Rightarrow \forall \lambda_1 \in \theta_1, \lambda_2 \in \theta_2 : f'(\lambda_1) \neq f'(\lambda_2) & \text{since } f'(\lambda_1) \in \beta, f'(\lambda_2) = \gamma \text{ and } \gamma \notin \beta \\
\gamma \in \beta \;\Rightarrow\; \forall \lambda_2 \in \theta_2 : f'(\lambda_2) \in Indef(\beta \uplus \gamma) & \text{since } \langle \sigma, Any \rangle \text{ is indefinite} \\
\forall \lambda_1, \lambda_2 \in \theta_2 : \lambda_1 \neq \lambda_2 \Rightarrow \gamma \text{ is indefinite} & \text{by Definition 7.}
\end{array}
$$

3. $Def(\beta) \subseteq codomain(f)$ By Definition 18

$$
\begin{array}{l}
\gamma \notin \beta \;\Rightarrow\; [\mathtt{Def}(\beta \uplus \gamma) \subseteq \{\gamma\} \cup \mathtt{Def}(\beta) \subseteq \{\gamma\} \cup codomain(f) = codomain(f')] \\
\gamma \in \beta \;\Rightarrow\; [\mathtt{Def}(\beta \uplus \gamma) = \mathtt{Def}(\beta) \subseteq codomain(f) \subseteq codomain(f')].
\end{array}
$$

□

Theorem 38 Let β_1 and β_2 are abstract stores and θ be a store.

$$\theta \in Cc(\beta_1) \vee \theta \in Cc(\beta_2) \;\Rightarrow\; \theta \in Cc(\mathtt{UNION}(\beta_1, \beta_2)).$$

Proof We only show that $\theta \in Cc(\beta_1) \Rightarrow \theta \in Cc(\mathtt{UNION}(\beta_1, \beta_2))$. The case $\theta \in Cc(\beta_2) \Rightarrow \theta \in Cc(\mathtt{UNION}(\beta_1, \beta_2))$ is similar. The proof is by induction on $|\beta_1| + |\beta_2|$. The basic case is trivial. Assume that the result holds for $|\beta_1| + |\beta_2| \leq n$. We show that it holds for $n+1$. Let $\theta \in Cc(\beta_1)$ and consider the various cases.

- *case* $2i$*:* We have that $\beta_1 = \{\gamma\} \cup \beta_1'$ $(\gamma \notin \beta_1')$ and $\gamma = \langle \sigma, One \rangle$. By definition of the concretization function, we have that $\theta = \theta_1 \uplus \theta_2$, $\theta_1 \in Cc(\gamma)$, and $\theta_2 \in Cc(\beta_1')$. By hypothesis, we have that $\theta_2 \in Cc(\mathtt{UNION}(\beta_1', \beta_2))$. We also have that $\theta_1 \in Cc(\{\langle \sigma, ZeroOrOne \rangle\})$ by monotonicity of Cc. Hence, by Lemma 33, we have $\theta_1 \uplus \theta_2 \in Cc(\mathtt{UNION}(\beta_1', \beta_2)) \uplus \{\langle \sigma, ZeroOrOne \rangle\}$.

- *case* $2ii$*:* By hypothesis, we have that $\theta \in Cc(\mathtt{UNION}(\beta_1, \beta_2'))$. By definition of the concretization function, we have that $\emptyset \in Cc(\{\langle \sigma, ZeroOrOne \rangle\})$. Hence, by Lemma 33,

$$\theta \in Cc(\mathtt{UNION}(\beta_1, \beta_2')) \uplus \{\langle \sigma, ZeroOrOne \rangle\}.$$

- *case* $3i$*:* Similar to case $2i$.

- *case* $3ii$*:* Similar to case $2ii$.

- *case* $4i$*:* We have that $\beta_1 = \{\gamma\} \cup \beta_1'$ $(\gamma \notin \beta_1')$ and $\gamma = \langle s_0 \; op \; \sum_{i=1}^n s_i x_i, \mu \rangle$. By definition of the concretization function, we have that $\theta = \theta_1 \uplus \theta_2$, $\theta_1 \in Cc(\gamma)$, and $\theta_2 \in Cc(\beta_1')$. By hypothesis, we have that $\theta_2 \in Cc(\mathtt{UNION}(\beta_1', \beta_2'))$. By monotonicity of Cc, we have that $\theta_1 \in Cc(\{\langle s_0 \sqcup s_0' \; op \; \sum_{i=1}^n s_i \sqcup s_i' x_i, \mu \sqcup \mu' \rangle\})$. The result follows from Lemma 33.

□

Lemma 40 Let θ be a constraint store, let β be an abstract store, and $v \in \mathcal{N}$. We have

$$\left.\begin{array}{l}
\mathtt{Csplit}(\theta, v) = \langle \theta^0, \theta^+, \theta^- \rangle \\
\theta \in Cc(\beta) \\
\mathtt{Asplit}(\beta, v) = \langle \beta^0, \beta^+, \beta^- \rangle
\end{array}\right\} \Rightarrow \theta^0 \in Cc(\beta^0) \;\wedge\; \theta^+ \in Cc(\beta^+) \;\wedge\; \theta^- \in Cc(\beta^-).$$

Proof By induction on $|\beta|$. The basic case is obvious. Assume that the result holds for $|\beta| \leq n$. We show that it holds for $|\beta| = n+1$. Let $\beta = \{\gamma\} \cup \beta_2 \quad (\gamma \notin \beta_2)$ and $|\beta_2| = n$. Let $\theta \in Cc(\beta)$. Then, $\theta = \theta_1 \uplus \theta_2$ with $\theta_1 \in Cc(\gamma)$ & $\theta_2 \in Cc(\beta_2)$. By the induction hypothesis, we know that

$$\theta_2^0 \in Cc(\beta_2^0) \ \wedge \ \theta_2^+ \in Cc(\beta_2^+) \ \wedge \ \theta_2^- \in Cc(\beta_2^-)$$

where

$$\texttt{Csplit}(\theta_2, v) = \langle \theta_2^0, \theta_2^+, \theta_2^- \rangle \text{ and } \texttt{Asplit}(\beta_2, v) = \langle \beta_2^0, \beta_2^+, \beta_2^- \rangle.$$

We show that $\theta^0 \in Cc(\beta^0)$. The other cases are similar. Consider $\gamma = \langle \sigma, \mu \rangle$. If $\sigma[v] \in \{\oplus, \ominus\}$, then θ_1 does not contain any constraint λ such that $\lambda[v] = 0$ by definition of the concretization. Hence, $\theta^0 = \theta_2^0$ and the result follows from the fact that $\beta^0 = \beta_2^0$.

If $\sigma[v] = 0$, then θ_1 contains only constraints such that $\lambda[v] = 0$, $\theta^0 = \theta_2^0 \uplus \theta_1$ and $\beta^0 = \beta_2^0 \uplus \gamma$. The result follows by Theorem 31.

If $\sigma[v] = \top$, then θ_1 contains a subset of constraints θ_1^0 which contains only constraints such that $\lambda[v] = 0$, $\theta^0 = \theta_2^0 \uplus \theta_2^1$ and $\beta^0 = \beta_2^0 \uplus \gamma$. We have that $\theta_1^0 \in Cc(\gamma^0)$ since all elements of θ_1^0 have a zero coefficient for x_v and since $\mu \sqsubseteq \mu \sqcup \mathit{ZeroOrOne}$. The result follows by Theorem 31. □

Lemma 45 [Gauss Step] Let $v \in \mathcal{N}$, σ be an abstract constraint such that $\sigma[v] \in \{\oplus, \ominus\}$ and $op(\sigma)$ is '=' and β be an abstract store such that, for all $\langle \sigma', \mu' \rangle \in \beta$, $\sigma'[v] \in \{\oplus, \ominus\} \ \wedge \ \sigma'[v] \neq \sigma[v]$. We have

$$\left.\begin{array}{l} \theta \in Cc(\beta) \\ \lambda \in Cc(\sigma) \end{array}\right\} \Rightarrow \texttt{Cgauss\_step}(\theta, \lambda, v) \in Cc(\texttt{Agauss\_step}(\beta, \sigma, v)).$$

Proof By induction on $|\beta|$. The basic case is obvious. Assume that the result holds for $|\beta| = n$. We show that it holds for $n+1$. Let $\beta = \{\gamma'\} \cup \beta' \quad (\gamma' \notin \beta')$ and $|\beta'| = n$. Let $\theta \in Cc(\beta)$. Then, $\theta = \theta_1^i \uplus \theta_2^i$ with $\theta_1^i \in Cc(\gamma')$ & $\theta_2^i \in Cc(\beta')$. We have

$$\begin{array}{lcl} \lambda & \in & Cc(\sigma) \\ \theta_1^i & \in & Cc(\gamma') \\ \theta_2^i & \in & Cc(\beta') \\ \theta_1 & = & \texttt{Cgauss\_step}(\theta_1^i, \lambda, v) \\ \theta_2 & = & \texttt{Cgauss\_step}(\theta_2^i, \lambda, v) \\ \beta_1 & = & \{\langle \texttt{Acombine}(\sigma', \sigma, v), \mu' \rangle\} \\ \beta_2 & = & \texttt{Agauss\_step}(\beta', \sigma, \mu) \end{array}$$

By hypothesis, we have

$$\theta_2 \in Cc(\beta_2).$$

By Lemma 39, the definition of the concretization function for abstract constraints with multiplicity, and the fact that there are as many constraints in θ_1 as in θ_1^i, we have

$$\theta_1 \in Cc(\beta_1).$$

The result follows from Lemma 33. □

Lemma 41 Let $v \in \mathcal{N}$, σ be an abstract constraint such that $op(\sigma)$ is '=' and $\sigma[v] = \oplus$, and β be an abstract store. We have

$$\theta \in Cc(\beta) \wedge \lambda \in Cc(\sigma) \Rightarrow \texttt{Cgauss}(\theta, \lambda, v) \in Cc(\texttt{Agauss}(\beta, \sigma, v)).$$

Proof This is a direct consequence of the consistency of `Asplit` (Lemma 40) and of `Agauss_step` (Lemma 45). □

Lemma 46 [Fourier Step] Let $v \in \mathcal{N}$, let β^+ be an abstract store such that for all $\gamma \in \beta^+$, $\gamma = \langle \sigma, \mu \rangle$ and $\sigma[v] = \oplus$, let β^- be an abstract store such that for all $\gamma \in \beta^-$, $\gamma = \langle \sigma, \mu \rangle$ and $\sigma[v] = \ominus$. We have

$$\left.\begin{array}{l} \theta^+ \in Cc(\beta^+) \\ \theta^- \in Cc(\beta^-) \end{array}\right\} \Rightarrow \texttt{Cfourier\_step}(\theta^+, \theta^-, v) \in Cc(\texttt{Afourier\_step}(\beta^+, \beta^-, v)).$$

Proof Let β^+ be $\{\gamma_1^+, \ldots, \gamma_n^+\}$ and β^- be $\{\gamma_1^-, \ldots, \gamma_m^-\}$. Let $\{(\gamma_1^+, \gamma_1^-), \ldots, (\gamma_n^+, \gamma_m^-)\}$ be the pairs considered in the algorithm. Consider the pair (γ_i^+, γ_j^-) $(1 \leq i \leq n$ & $1 \leq j \leq m)$ such that $\gamma_i^+ = \langle \sigma_i^+ , \mu_i^+ \rangle$ and $\gamma_j^- = \langle \sigma_j^- , \mu_j^- \rangle$ and consider $\theta^+ \in Cc(\gamma_i^+)$ and $\theta^- \in Cc(\gamma_j^-)$. By Lemma 39, we have that any pair of constraints (λ^+, λ^-) from (γ_i^+, γ_j^+) satisfy

$$\texttt{Ccombine}(\lambda^+, \lambda^-, v) \in Cc(\texttt{Acombine}(\sigma_i^+, \sigma_j^-, v)).$$

Moreover, the number of constraints in $\texttt{Cfourier\_step}(\theta^+, \theta^-)$ is $|\theta^+| \times |\theta^-|$. The upper bound operation on multiplicities approximates this product, since $\mu_i^+ \sqcup \mu_j^-$ is *Any* as soon as one of them is *Any*, is *ZeroOrOne* as soon as none of them is *Any* and one of them is *ZeroOrOne*, and is *One* otherwise. As a consequence,

$$\theta^+ \in Cc(\gamma_i^+) \;\&\; \theta^- \in Cc(\gamma_j^-) \;\Rightarrow\; \texttt{Cfourier\_step}(\theta^+, \theta^-) \in Cc(\langle \texttt{Acombine}(\sigma_i^+, \sigma_j^-, v), \mu_i^+ \sqcup \mu_j^- \rangle)$$

The result follows from Lemma 33. □

Lemma 42 Let $v \in \mathcal{N}$, θ be a store, and β be an abstract store. We have

$$\theta \in Cc(\beta) \Rightarrow \texttt{Cfourier}(\theta, v) \in Cc(\texttt{Afourier}(\beta, v)).$$

Proof This follows from the consistency of `Asplit` (Lemma 40) and of `Afourier_step` (Lemma 46) and from the definition of the concretization which specifies that, when γ is of the form $\langle \sigma, \mu \rangle$ with $\mu \in \{One, ZeroOrOne\}$, any $\theta \in Cc(\gamma)$ is of cardinality at most one. □

Theorem 43 Let $v \in \mathcal{N}$, θ be a store and β be an abstract store.

$$\theta \in Cc(\beta) \;\Rightarrow\; \text{there exists } \theta' \text{ such that } \theta' \leftrightarrow \exists v \theta \;\wedge\; \theta' \in Cc(Aproject(\beta, v)).$$

Proof Let β be $\{\gamma\} \cup \beta'$ $(\gamma \notin \beta')$ with $\gamma = \langle \sigma, \mu \rangle$ and $\theta \in Cc(\beta)$. If $\mu = One$ and op(σ) is '=' and $\sigma[v] = \oplus$, then $\theta = \{\lambda\} \uplus \theta'$, $\lambda \in Cc(\gamma)$ and $\theta' \in Cc(\beta')$. By definition, of the concretization, $\lambda[v] \geq 0$ and a concrete projection algorithm can use Gaussian elimination on λ. By Lemma 41, we have

$$\texttt{Cgauss}(\theta', \lambda, v) \in Cc(\texttt{Agauss}(\beta', \sigma, v))$$

and the result follows. The case $\mu = One \wedge$ op(σ) is '=' $\wedge\; \sigma[v] = \ominus$ is similar. Finally, a concrete projection algorithm can always use Fourier elimination and the result follows from Lemma 42. □