ABSTRACT DOMAINS FOR REORDERING CLP($\Re_{Lin}$) PROGRAMS

V. Ramachandran, P. Van Hentenryck, and A. Cortesi

▷ In order to address the multi-directional nature of constraint logic programs, recent optimizing compilers generate several versions of a procedure and optimize them independently. Reordering, i.e., moving constraints towards the end of a clause, plays a fundamental role in this optimization: it may lead to significant improvements in performance by bypassing the constraint solver entirely. This paper focuses on CLP over linear real constraints, and studies two abstract domains, i.e., `LSign` and `LInt`, which can be used to decide at compile time when constraints can be safely reordered. The domain `LSign` was originally proposed by Marriott and Stuckey. Its fundamental ideas consist of abstracting coefficients by signs and of keeping multiplicity information on constraints. `LInt` is a new, and infinite, domain which is similar in nature to `LSign`, except that signs are replaced by intervals of rational numbers. A comprehensive description of the two domains is given, together with some very preliminary evidence showing that the domains are precise enough to perform the intended optimizations on small programs. ◁

1. Introduction

Constraint logic programming (`CLP`) [19] is a generalization of logic programming where constraint solving over a suitable domain replaces unification as the basic operation of the language. Many CLP languages have been defined in the last decade on computation domains such as linear real constraints (e.g., [20, 38, 4]), integers (e.g., [37]), Booleans (e.g., [3, 4]), and nonlinear real constraints (e.g., [30]).

Address correspondence to Agostino Cortesi, Università Ca' Foscari, via Torino 155, 30173 Venezia, Italy, tel +39-41-2908.428, fax +39-41-2908.419, email `cortesi@dsi.unive.it`.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

CLP languages offer many challenges for optimizing compilers since constraint solving, its basic operation, may be quite expensive in general. It is thus natural to try to specialize it whenever appropriate. The optimization problem is also exacerbated by the fact that CLP programs can be multi-directional (i.e., they may be used with different combinations of input/output parameters). Hence, a given program may be appropriate for some uses, while being 'sub-optimal" for some others.

This paper focuses on CLP($\Re_{Lin}$) programs [20, 39, 35], i.e. on constraint logic programs over linear real constraints, and the approach to CLP optimization proposed in [27]. This methodology consists of refining constraints into tests and assignments, of removing redundant constraints (i.e., constraints which are implied by the constraint store), and of reordering constraints to maximize refinements and removals, while preserving the same search space. The optimizations are specific to the domain $\Re$ of real constraints but similar optimizations can be applied to other domains as well. These optimizations are global in nature, and require information about the macro properties of the program in order to be automated successfully.

Among these various optimizations, reordering is probably the most fundamental. The need for reordering comes from the multi-directional nature of CLP programs. A traditional technique to address this issue in recent optimizing compilers (e.g., [31, 23, 24, 32]) consists of generating one version of each procedure for each use and of specializing each version independently. Reordering is fundamental in this optimization: it consists of moving constraints toward the end of the clause so that they can be turned into tests or assignments. Of course, reordering should only be applied when postponing the constraints does not change the search space explored by the program (and thus does not change the termination behaviour of the program). In other words, reordering makes it possible to bypass the constraint solver while preserving the same search space, thus improving the performance of the program in many cases.

This paper studies two abstract domains, LSign and LInt, which can be used to decide when to reorder CLP programs. LSign is an abstract domain originally proposed in [28] and further studied in [33]. Its fundamental ideas consist of abstracting coefficients by signs and of keeping multiplicity information on constraints. LInt is a new, and infinite, domain which is similar in nature to LSign, except that signs are replaced by intervals of rational numbers. Note that other domains have been proposed for the analysis of CLP($\Re_{Lin}$) programs (e.g., [10, 11, 14, 15]) and these domains are potentially useful for reordering as well. However, as acknowledged in [15], they are less precise than LSign and, as shown later in this paper, applying LSign itself is not easy in practice. Hence, the study of LSign and LInt gives another entry point on the analysis of CLP($\Re_{Lin}$) programs.

The focus of the paper is to give a comprehensive presentation of these domains together with their correctness proofs and their application to reordering. As shown in [33], LSign raises some interesting issues and obtaining correct abstract operations is non trivial. In addition, as shown later on in this paper, applying LSign in a naive way in an optimizing compiler is unsatisfactory: "obvious" optimizations cannot be performed automatically if no special care is taken. The paper illustrates these problems, proposes solutions, and provides some preliminary empirical evidence that the resulting algorithms are precise enough to optimize small programs. This paper does not discuss the reordering algorithm and its correctness proof: they are discussed in [36] and deserve an independent treatment.

As far as `LSign` is concerned, the paper redefines the domain of [28] to fix some loose ends and to simplify the correctness proofs. It proposes a new ordering capturing the intended meaning of [28] and shows that two `LSign` descriptions can be ordered in polynomial time. It also proposes a more precise algorithm for projection. Finally, the paper discusses how `LSign` can be used to detect the conditional satisfiability of constraint stores, an operation which is fundamental to reorder CLP($\Re_{Lin}$) programs. This operation, which was never discussed previously, raises some subtle practical issues which are studied at length.

As far as `LInt` is concerned, the paper contains the first presentation of the domain, and proposes a practical and precise widening operator. The operator has the original property of using `LSign` to guide the widening process. The paper also contains some preliminary empirical evidence of the overhead of `LInt` over `LSign`.

The rest of this paper is organized as follows. The next section explains why reordering is so fundamental in CLP($\Re_{Lin}$) optimizations. Section 3 describes the concrete objects and operations. Section 4 and Section 5 give a detailed presentation of the abstract domains `LSign` and `LInt`, respectively. Section 6 presents the preliminaries experimental results. Section 7 concludes the paper. The proofs of the results can be found in the technical report version of this paper [34]. Only sketches of these proofs are given here for space reasons.

2. Why Reordering is Important?

To motivate the paper, it is interesting to study an example of optimization in some detail. Consider the following CLP($\Re_{Lin}$) program [20] which relates various parameters in a mortgage computation. This program will be used as running example in the paper.

Example 2.1.

```
mg(P,T,R,B) :- T = 0, B = P.
mg(P,T,R,B) :- T > 0,P ≥ 0,P1 = P*1.01 - R,T1 = T - 1,mg(P1,T1,R,B).
```

The predicate `mg` relates the principal (`P`), number of monthly installments (`T`), monthly repayment (`R`) and final balance (`B`) of a mortgage that has a fixed monthly interest rate of 1%. The most interesting feature of this program is its multi-directionality. For example, the query

```
:- mg(P,4,200,0).
```

computes the principal of a mortgage having 4 installments of 200 units each, with a final balance of 0. The answer is `P = 780.39`. The program can also be used to compute the monthly repayment given the principal. For example, the query

```
:- mg(800,4,R,0).
```

gives the answer `R = 205.02`. An even more interesting query is to find out the repayments such that each repayment is less than 200 units and there are at most 6 installments. This is given by

```
:- R < 200, T ≤ 6, mg(800,T,R,0).
```

and gives the answers `T = 5, R = 164.83` and `T = 6, R = 138.04`.

In general, the predicate `mg` above can be used to compute the (complicated) relations between any of the parameters in the mortgage computation because of the general constraint solver embedded in the language. For some of the above uses, however, this generality is not needed and it is possible to generate a completely deterministic program which does not even resort to constraint solving. In the rest of this section, we present how this is achieved by the first pass of the optimizing compiler [36]. The first pass of the compiler may be viewed as a source to source transformation, transforming the source program into another (richer) source program which may contain, not only constraints, but also assignments and tests as basic operations. The source to source transformation is organized in three main phases: reordering, removal, and refinement.

Consider the case where `mg` is used with `P` and `R` fixed (i.e., they are constrained to take a value) and `T` and `B` are unconstrained. The first step of the compiler tries to move constraints to a place in the clause where, roughly speaking, they can be specialized into tests or into assignments. More precisely, an inequality is moved to a place where all its variables are fixed at runtime, while an equation is moved to a place where all its variables or all its variables but one are fixed at runtime. In reordering goals in a clause, the compiler should make sure that the search space explored by the program is preserved in order to guarantee termination and to avoid significant slowdowns. On our running example, the compiler postpones `T > 0` in the second clause until after the recursive call. Informally speaking, this is possible due to the fact that `T > 0` is always consistent with the constraint stores occurring in the recursive call to `mg` and, consequently, `T > 0` does not prune the search space. This information can be formally derived through an `LSign` or `LInt` analysis. Note also that, when postponed until after the recursive call, `T > 0` may be specialized into a test, since `T` is fixed. Similarly, `T1 = T - 1` can be moved until after the recursive call, since `T` is now unconstrained before the recursive call and hence it cannot prune the search space. The resulting program becomes:

```
mg(P,T,R,B) :- T = 0, B = P.
mg(P,T,R,B) :- P ≥ 0, P1 = P*1.01 - R, mg(P1,T1,R,B), T = T1 + 1, T > 0.
```

Once the program is reordered, other optimizations can take place. A second step consists of detecting redundant constraints (i.e., constraints implied by the constraint store each time they are selected). These constraints can be removed, since they do not add information to the constraint store. In our running example, this is the case of `T > 0` after reordering, since informally speaking, the second argument is assigned to zero in the first clause and incremented by one in the recursive clause.[1] Once again, this fact can be proven formally through abstract interpretation using the domain `LSign` or `LInt` or other domains such as the convex hull domain of [8]. The resulting program is as follows:

```
mg(P,T,R,B) :- T = 0, B = P.
mg(P,T,R,B) :- P ≥ 0, P1 = P*1.01 - R, mg(P1,T1,R,B), T = T1 + 1.
```

Finally, the last step of the source to source transformation specializes constraints into tests and assignments whenever possible. A constraint can be specialized into a test if the compiler shows that, at runtime, all its variables are fixed. An equation `Var = Exp` can be transformed into an assignment if the compiler shows that, at runtime, `Var` is unconstrained and `Exp` is a fixed expression. In our running example,

[1] Note that this constraint is useful for other uses of the program.

it can be shown that, after reordering and removal, `T` and `B` are unconstrained in all calls to `mg`. Once again, this information can be deduced from domains such as `LSign` and `LInt`. Also `P` and `R` are constrained to take a value in all calls to `mg`. As a consequence, the constraints in the first clause becomes assignments, while the second clause has two assignments and a test.

```
mg(P,T,R,B) :- T := 0, B := P.
mg(P,T,R,B) :- P ?≥ 0, P1 := P*1.01 - R, mg(P1,T1,R,B), T := T1 + 1.
```

It is interesting to observe that the resulting program does not invoke the constraint solver. It is essentially a Prolog program enhanced with a rational arithmetic component. As a consequence, traditional Prolog transformations and optimizations can now be applied. For instance, the techniques of [9] can be used to transform our final program into a tail-recursive program. Similarly, efficient instructions can be generated for the tests and assignments [21].

3. The Concrete Domain

Abstract interpretation [6] is a general methodology to design static analyses of programs. The basic intuition is to infer some properties of a program, not by executing it on its traditional computation domain, but rather on an abstract domain. The abstract domain should of course be designed to approximate the properties of interest with reasonable precision and with reasonable computer resources. Traditionally, an abstract interpretation is constructed in four steps: a semantics of the language is defined, called the standard semantics; the standard semantics is transformed into a collecting (concrete) semantics [2]; the collecting semantics is abstracted into an abstract semantics; the abstract semantics (or a subset of it) is computed.

This paper follows this methodology for the analysis of CLP programs. The paper does not describe each of these steps in detail, since the focus here is on the abstract domains. It suffices to say that the concrete semantics for CLP is a natural extension to CLP of the logic programming semantics presented in many papers. It captures the top-down execution of CLP programs using a left-to-right computation rule, and ignores the clause selection rule. The concrete semantics is defined for normalized CLP programs and it manipulates sets of constraint stores (multisets of linear constraints). The rest of this section formally defines the concrete concepts necessary for this paper. Readers interested in a comprehensive presentation of the various steps should refer to [36]. See also [11, 14, 15, 24, 23] for applications of static analysis to CLP.

Programs. A $\mathtt{CLP}(\Re_{Lin})$ program is a (possibly empty) sequence of clauses in which each clause has a head and a body. A head is an atom, i.e. an expression of the form $\mathtt{p}(\mathtt{t}_1,\ldots,\mathtt{t}_n)$ where $\mathtt{t}_1,\ldots,\mathtt{t}_n$ are terms. A term is a variable (e.g. `X`) or a rational number. A body is either `true` (the empty body), a goal (procedure call), a constraint (constraint solving) or a sequence of these. In the following, variables are denoted by uppercase letters, constraints by the letter c, possibly subscripted or superscripted. The constraints of $\mathtt{CLP}(\Re_{Lin})$ can be specified as follows:

[2] This second step is needed in general because the analysis aims at giving some information about the results of a computation for a set of input values, not a single input value.

Definition 3.1. Let t_1 and t_2 be two linear expressions constructed with variables, rational numbers, and the operations $+, *, -$ and $/$. A constraint is a relation $t_1\ \delta\ t_2$ with $\delta \in \{>, \geq, =, \neq, \leq, <\}$.

It is convenient to normalize `CLP(`$\Re_{Lin}$`)` programs in order to simplify the analysis. Normalized programs are defined in terms of sets P_i $(i \geq 0)$ representing the set of predicate symbols of arity i and of an infinite sequence of variables $x_1, x_2, \ldots$. A *normalized program* is a (possibly empty) sequence of clauses, in which each clause has a head and a body. The head of a clause has the form $p(x_1, \ldots, x_n)$ where $p \in P_n$. The body of a clause is a (possibly empty) sequence of literals where each literal is either a predicate call or a linear constraint. A predicate call has the form $p(x_{i_1}, \ldots, x_{i_m})$ where $p \in P_m$ and the variables $x_{i_1}, \ldots, x_{i_m}$ are all distinct. Linear constraints are defined below. Any given `CLP(`$\Re_{Lin}$`)` program may be transformed into an equivalent normalized program by simple rewriting rules [2].

Concrete Objects. The concrete objects manipulated by our concrete semantics are *linear constraints and multisets of linear constraints.* Given a set of variables $D = \{x_1, \ldots, x_n\}$, a linear constraint over D is an expression of the form $c_0\ \delta\ \sum_{i=1}^{n} c_i x_i$, where c_i are rational numbers and $\delta \in \{=, <, \leq, >, \geq, \neq\}$. 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 [3]. In the following, we denote multiset union by $\uplus$, linear constraints by the letter λ, constraint stores by the letter θ and sets of constraint stores by the letter Θ, all possibly subscripted. If λ is a linear constraint $c_0\ \delta\ \sum_{i=1}^{n} c_i x_i$, $\lambda[i]$ denotes the coefficient c_i, and $op(\lambda)$ denotes the operator δ. We denote the set of variables $\{x_1, \ldots, x_n\}$ by D_n for any n. If θ is a constraint store over the set of variables D, $dom(\theta) = D$. The projection of a constraint store θ on the set of variables D is denoted $\theta_{/D}$. If θ is a constraint store over $D = \{x_1, \ldots, x_n\}$ and $x \in D$, θ_x^+, θ_x^- and θ_x^0 represent all constraints λ in θ whose coefficient for variable x is respectively strictly positive, strictly negative, and zero. We use $\mathtt{Var}(\theta)$ to denote the set of variables with non-zero coefficients in θ. We also denote by $\mathcal{I}$ the set $\{i \mid x_i \in D\}$.

Concrete Operations. The concrete semantics of CLP programs requires three main concrete operations: addition of a constraint to a constraint store, projection of a constraint store on some variables, and union of constraint stores. We now define these concrete operations in more detail.

Upper bound. Let $\Theta_1, \ldots, \Theta_m$ be sets of constraint stores on D_n. $\mathtt{UNION}(\Theta_1, \ldots, \Theta_m)$ returns a set of constraint stores that represents all the constraint stores in Θ_i $(1 \leq i \leq m)$. Operation `UNION` is used to compute the result of a predicate given the results of its individual clauses. It is specified as follows.

Specification 1. Let $\Theta_1, \ldots, \Theta_m$ be sets of constraint stores on D_n. Then

$$\mathtt{UNION}(\Theta_1, \ldots, \Theta_m) = \Theta_1\ \cup \ldots \cup\ \Theta_m.$$

[3] Observe that this is a partial order, and that constraint stores, not comparable by the order relation, can be equivalent. This is harmless, as the analysis is always conservative, but it may introduce some redundancies.

Addition of a constraint. Let Θ be a set of constraint stores on D_n and λ be a constraint on D_n. $\texttt{AI\_ADD}(\lambda, \Theta)$ returns a set of constraint stores that represents the result of adding the constraint λ to each of the constraint stores in Θ. Operation AI_ADD is used when a constraint is encountered in the program.

Specification 2. Let Θ be a set of constraint stores on D_n and λ be a constraint on D_n.

$$\texttt{AI\_ADD}(\lambda, \Theta) = \{\theta \uplus \{\lambda\} \mid \theta \in \Theta\}.$$

Projection of a constraint store. Let Θ be a set of constraint stores on variables D, and let V be a subset of these variables. $\texttt{PROJECT\_SET}(\Theta, V)$ returns the set of constraint stores obtained by projecting Θ on $R = D \setminus V$. It is specified as follows.

Specification 3. Let Θ be a set of constraint stores on D, $V \subseteq D$, and $R = D \setminus V$. Then

$$\texttt{PROJECT\_SET}(\Theta, V) = \{\theta_{/R} \mid \theta \in \Theta\}.$$

This operation is used in many phases in the concrete semantics, e.g. just before a procedure call to project the constraint stores on the call variables, or at the exit of a clause to project the constraint stores on the head variables.

4. The Abstract Domain LSign

In this section, we present the abstract domain LSign. First, Section 4.1 presents the abstract domain used to approximate linear constraints and multisets of linear constraints. Sections 4.2 and 4.3 contain the operations and applications of the domain. This is followed by a brief presentation of the power domain in Section 4.4.

4.1. The Domain

As mentioned in the introduction, the domain was first introduced in [28] and then revisited in [33]. The presentation here is based on [33] which clearly separates multiplicity information from the abstract constraints.[4]The 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, [28] 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 $D = \{x_1, \ldots, x_n\}$. An *operator* is an element of $\texttt{Op} = \{<, \leq, =\}$. Operators are denoted by the letter δ.

Definition 4.1. [Signs] A *sign* is an element of $\texttt{Sign} = \{0, \oplus, \ominus, \top\}$. Sign is ordered by $s_1 \sqsubseteq s_2 \Leftrightarrow (s_1 = s_2) \vee (s_2 = \top)$. The (monotone) concretization function $Cc : \texttt{Sign} \to 2^{\Re}$ is defined as

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

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

Definition 4.2. [Abstract Constraints] An *abstract constraint* over D is an expression of the form $s_0\ \delta\ \sum_{i=1}^{n} s_i x_i$ where s_i is a sign and δ is an operator. The set of abstract constraints over D is denoted by A_D and is partially ordered by

$$s_0\ \delta\ \sum_{i=1}^{n} s_i x_i \sqsubseteq s'_0\ \delta\ \sum_{i=1}^{n} s'_i x'_i \Leftrightarrow \bigwedge_{i=0}^{n} (s_i \sqsubseteq s'_i).$$

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

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

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

Example 4.1. The abstract constraint $\top = \oplus\mathtt{P} + \oplus\mathtt{R}$ represents both the constraint $3 = \mathtt{P} + \mathtt{R}$ and $-3 = 2\mathtt{P} + 5\mathtt{R}$ but not the constraint $3 = -\mathtt{P} + \mathtt{R}$.

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.

Definition 4.3. [Multiplicities] A *multiplicity* is an element of $\mathtt{Mult} = \{\mathtt{One}, \mathtt{ZeroOrOne}, \mathtt{Any}\}$. `Mult` is ordered by $\mathtt{One} \sqsubseteq \mathtt{ZeroOrOne} \sqsubseteq \mathtt{Any}$. Multiplicities are denoted by the letter μ, possibly subscripted.

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

Definition 4.4. [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 (monotone) concretization function $Cc : AM_D \to CS_D$ is defined as

$$\begin{aligned} &Cc(\langle \sigma, One \rangle) = \{\ \{\lambda\} \mid \lambda \in Cc(\sigma)\ \} \\ &Cc(\langle \sigma, \mathtt{ZeroOrOne} \rangle) = \{\emptyset\} \cup \{\ \{\lambda\} \mid \lambda \in Cc(\sigma)\ \} \\ &Cc(\langle \sigma, \mathtt{Any} \rangle) = \{\emptyset\} \cup \{\ \{\lambda_1, \ldots, \lambda_m\} \mid m \ge 1 \wedge \lambda_i \in Cc(\sigma)\ (1 \le i \le m)\ \}. \end{aligned}$$

Abstract constraints with multiplicities [5] 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 4.2. The abstract constraint with multiplicity $\langle \top = \oplus\mathtt{P} + \oplus\mathtt{R}, \mathtt{One} \rangle$ represents only multisets of size 1, e.g., $\{3 = \mathtt{P} + \mathtt{R}\}$. $\langle 0 \le \oplus\mathtt{P} + \ominus\mathtt{R}, \mathtt{Any} \rangle$ represents

[5] [28] contains one additional 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 [28].

multisets of any size, e.g., $\emptyset$, $\{0 \le 3\mathtt{P} - \mathtt{R}\}$ and $\{0 \le 3\mathtt{P} - \mathtt{R}, 0 \le 2\mathtt{P} - 3\mathtt{R}\}$.

We are now in position to define the abstract stores of the domain, which abstract the constraint stores in the concrete domain.

Definition 4.5. [Abstract Stores] An *abstract store* over D is a set β of abstract constraints with multiplicities. The set of abstract stores is denoted by AS_D. The concretization function $Cc : AS_D \to CS_D$ is defined in two stages. The first captures the syntactic form of the abstract store.

$$Cc_i(\emptyset) = \{\emptyset\}$$
$$Cc_i(\{\gamma\} \cup \beta) = \{\ \theta_1 \uplus \theta_2 \mid \theta_1 \in Cc(\gamma) \wedge \theta_2 \in Cc_i(\beta)\ \}\ \gamma \notin \beta.$$

The second captures the extension to equivalence classes in the concrete domain.

$$Cc(\beta) = \{\ \theta \mid \theta \leftrightarrow \theta_i \wedge \theta_i \in Cc_i(\beta)\ \}$$

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

Example 4.3. The abstract store $\beta = \{\langle \top = \oplus\mathtt{P} + \oplus\mathtt{R}, \mathtt{One}\rangle,\ \langle 0 \le \oplus\mathtt{P} + \ominus\mathtt{R}, \mathtt{Any}\rangle\}$ represents constraint stores with at least one constraint, and their equivalence classes. For example $\{3 = \mathtt{P} + \mathtt{R}\} \in Cc_i(\beta)$ and $\{3 = \mathtt{P} + \mathtt{R},\ 0 \le 2\mathtt{P} - 3\mathtt{R}\} \in Cc_i(\beta)$. Further, $\{3 = \mathtt{P} + \mathtt{R},\ 9 \le 5\mathtt{P}\} \in Cc(\beta)$ because $\{3 = \mathtt{P} + \mathtt{R},\ 9 \le 5\mathtt{P}\} \leftrightarrow \{3 = \mathtt{P} + \mathtt{R},\ 0 \le 2\mathtt{P} - 3\mathtt{R}\}$.

It remains to define the ordering on abstract stores. Our goal is to make sure that $\beta_1 \sqsubseteq \beta_2$ 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 4.6. [Definite, Possible, and Indefinite Constraints] Let γ be an abstract constraint with multiplicity. γ is a *definite* abstract constraint iff $mult(\gamma) = \mathtt{One}$. γ is a *possible* abstract constraint iff $mult(\gamma) = \mathtt{ZeroOrOne}$. It is an *indefinite* abstract constraint otherwise. The *definite portion* of β, denoted by $\mathtt{Def}(\beta)$, is the set of definite constraints of β. The *possible portion* of β, denoted by $\mathtt{Pos}(\beta)$, is the set of possible constraints of β. The *indefinite portion* of β, denoted by $\mathtt{Indef}(\beta)$, is the set of indefinite constraints of β.

Definition 4.7. [Ordering Function] Let β_1 and β_2 be two abstract stores. An *ordering function* of β_1 to β_2 is a function $f : \beta_1 \to \beta_2$ satisfying

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

Definition 4.8. [Ordering on Abstract Stores] Let β_1 and β_2 be two abstract stores. $\beta_1 \sqsubseteq \beta_2$ if there exists an ordering function f of β_1 to β_2.

This definition of ordering indicates that we reason only on the syntactic form of the abstract stores. The first condition states that, for each abstract constraint with multiplicity in β_1, say γ, there exists an abstract constraint with multiplicity in β_2, say $f(\gamma)$, 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 [6].

Example 4.4. Consider the following abstract constraints with multiplicity.

$$\begin{array}{ll}\gamma_1 = \langle \oplus = \oplus \mathtt{P} + \oplus \mathtt{R}, \mathtt{One}\rangle & \gamma_4 = \langle \oplus = \oplus \mathtt{P} + \oplus \mathtt{R}, \mathtt{One}\rangle \\ \gamma_2 = \langle \oplus = \ominus \mathtt{P} + \oplus \mathtt{R}, \mathtt{One}\rangle & \gamma_5 = \langle \oplus = \ominus \mathtt{P} + \top \mathtt{R}, \mathtt{Any}\rangle \\ \gamma_3 = \langle \oplus = \ominus \mathtt{P} + \oplus \mathtt{R}, \mathtt{Any}\rangle & \gamma_6 = \langle \oplus = \oplus \mathtt{P} + \top \mathtt{R}, \mathtt{One}\rangle\rangle\end{array}$$

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

It can be proved that the ordering on abstract stores is transitive and reflexive, leading to the following theorem.

Theorem 4.1 Ordering Relation. $\sqsubseteq$: $AS_D \times AS_D$ is a pre-order.

We now turn to the first main result of this section: the monotonicity of the concretization function for abstract stores. 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. We define the notion of lifting function which is the counterpart of the ordering function for a pair (concrete store, abstract store).

Definition 4.9. [Lifting Function] Let β be an abstract store and θ a concrete constraint store. A *lifting function* of θ to β is a function $f : \theta \rightarrow \beta$ 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. $\mathtt{Def}(\beta) \subseteq range(f)$.

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

Lemma 4.1 Lifting Function Lemma. Let β be an abstract store and θ a concrete constraint store. $\theta \in Cc_i(\beta)$ if and only if there exists a lifting function f of θ to β.

Sketch of Proof: The proof proceeds by induction on $|\beta|$. The basic case is obvious. For the 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 $|\beta| = n+1$

[6] Notice that $\{\langle \oplus = \oplus x_1 + \oplus x_2, \mathtt{One}\rangle\}$ is smaller than $\{\langle \oplus = \oplus x_1 + \top x_2, \mathtt{One}\rangle\}$, and $\{\langle \oplus = \oplus x_1 + \ominus x_2, \mathtt{One}\rangle\}$ is smaller than $\{\langle \oplus = \oplus x_1 + \top x_2, \mathtt{One}\rangle\}$, but it is not true that $\{\langle \oplus = \oplus x_1 + \oplus x_2, \mathtt{One}\rangle, \langle \oplus = \oplus x_1 + \ominus x_2, \mathtt{One}\rangle\}$ is smaller than $\{\langle \oplus = \oplus x_1 + \top x_2, \mathtt{One}\rangle\}$. The latter holds in [28], but apparently it is not what was intended. This problem showed up when trying to prove the correctness of the abstract operations of `LSign` using the ordering defined in [28], and made it impossible, obviously.

and let β be $\{\gamma\} \cup \beta'$, where $\gamma \notin \beta'$ and $|\beta'| = n$. By Definition 5,

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

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

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

The proof consists of showing that f is a lifting function of θ to β.
($\Leftarrow$) Let f be a lifting function of θ to $\beta = \{\gamma\} \cup \beta'$. Let $\theta_1 = \{\ \lambda \mid \lambda \in \theta\ \wedge f(\lambda) = \gamma\}$. Then $\forall \lambda \in \theta_1 : \lambda \in Cc(cons(\gamma))$ by Definition 9 and $\theta_1 \in Cc(\gamma)$ by Definition 4. Let $\theta = \theta_1 \uplus \theta'$ and consider the function $f' : \theta' \to \beta'$ defined by $f'(\lambda) = f(\lambda)$ for all $\lambda \in \theta'$. The proof consists of showing that f' is a lifting function of θ' to β'. Then, by hypothesis, $\theta' \in Cc_i(\beta')$ and, by Definition 5, $\theta_1 \uplus \theta' \in Cc_i(\{\gamma\} \cup \beta')$.

□

Example 4.5. Consider the following abstract constraints with multiplicity

$$\gamma_1 = \langle \oplus = \oplus\mathtt{P} + \oplus\mathtt{R}, \mathtt{One}\rangle \qquad \gamma_2 = \langle \oplus = \oplus\mathtt{P} + \oplus\mathtt{R}, \mathtt{One}\rangle$$
$$\gamma_3 = \langle \oplus = \ominus\mathtt{P} + \top\mathtt{R}, \mathtt{Any}\rangle \qquad \gamma_4 = \langle \oplus = \oplus\mathtt{P} + \top\mathtt{R}, \mathtt{One}\rangle$$

and concrete constraints

$$\lambda_1 = 2 = 2\mathtt{P} + 3\mathtt{R} \qquad \lambda_2 = 3 = -2\mathtt{P} + 2\mathtt{R}$$
$$\lambda_3 = 1 = -3\mathtt{P} + 2\mathtt{R} \qquad \lambda_4 = 4 = 4\mathtt{P} + 3\mathtt{R}$$

$\{\lambda_1, \lambda_2\} \in Cc_i(\{\gamma_2, \gamma_3\})$ and the lifting function is defined by $f(\lambda_1) = \gamma_2$ and $f(\lambda_2) = \gamma_3$. Moreover, $\{\lambda_1, \lambda_2, \lambda_3\} \in Cc_i(\{\gamma_2, \gamma_3\})$ and the function is defined by $f(\lambda_1) = \gamma_2$, $f(\lambda_2) = \gamma_3$, and $f(\lambda_3) = \gamma_3$. Finally, $\{\lambda_3, \lambda_4\} \notin Cc_i(\{\gamma_1, \gamma_2, \gamma_4\})$, since there is no function that can cover all three of γ_1, γ_2 and γ_4 with only λ_3 and λ_4.

We are now in position to state the first main result of this section.

Theorem 4.2. [Monotonicity of Concretization Function w.r.t. Ordering Relation] If β_1 and β_2 are two abstract stores then
(i) $\beta_1 \sqsubseteq \beta_2 \Rightarrow Cc_i(\beta_1) \subseteq Cc_i(\beta_2)$.
(ii) $\beta_1 \sqsubseteq \beta_2 \Rightarrow Cc(\beta_1) \subseteq Cc(\beta_2)$.

Sketch of Proof: (i) Let $\beta_1 \sqsubseteq \beta_2$ and $\theta \in Cc_i(\beta_1)$. We need to show that $\theta \in Cc_i(\beta_2)$. By Lemma 4.1, there exists a lifting function f of θ to β_1. By Definition 8, there exists an ordering function g of β_1 to β_2. Consider the function $g \circ f$. The proof just shows that $g \circ f$ is a lifting function of θ to β_2.
(ii) Let $\beta_1 \sqsubseteq \beta_2$ and $\theta \in Cc(\beta_1)$. By definition of Cc, there exists $\theta_i \in Cc_i(\beta_1)$ s.t. $\theta \leftrightarrow \theta_i$. By part (i), $\theta_i \in Cc_i(\beta_2)$. Hence, by definition of Cc, $\theta \in Cc(\beta_2)$. □

For subsequent sketches of proof where the result is stated in two parts, the first part relating to Cc_i and the second part relating to Cc, we only give the sketch of proof for the first part. The extension for Cc follows the outline of the proof above.

4.2. 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.

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 4.10. [Bipartite Graph] A graph $G = (V, E)$ is bipartite if V can be partitioned into two sets V_1, 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 4.11. [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 $\mathtt{Def}(\beta_2)$, a vertex for each constraint in $\mathtt{Pos}(\beta_2)$, and $|\beta_1|$ vertices for each constraint in $\mathtt{Indef}(\beta_2)$, since these constraints can be used several times and the matching problem requires that each vertex appears at most 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 4.12. 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

$$
\begin{aligned}
V &= V_1 \cup V_2 \cup V_3 \cup V_4 \\
V_1 &= \{\gamma_1 \mid \gamma_1 \in \beta_1\} \\
V_2 &= \{\gamma_2 \mid \gamma_2 \in \mathtt{Def}(\beta_2)\} \\
V_3 &= \{\gamma_2 \mid \gamma_2 \in \mathtt{Pos}(\beta_2)\} \\
V_4 &= \{{\gamma_2}^{\gamma_1} \mid \gamma_2 \in \mathtt{Indef}(\beta_2) \wedge \gamma_1 \in \beta_1\}
\end{aligned}
$$

$$
\begin{aligned}
E &= E_2 \cup E_3 \cup E_4 \\
E_2 &= \{(\gamma_1, \gamma_2) \mid \gamma_1 \in V_1 \wedge \gamma_2 \in V_2 \wedge \gamma_1 \sqsubseteq \gamma_2\} \\
E_3 &= \{(\gamma_1, \gamma_2) \mid \gamma_1 \in V_1 \wedge \gamma_2 \in V_3 \wedge \gamma_1 \sqsubseteq \gamma_2\} \\
E_4 &= \{(\gamma_1, {\gamma_2}^{\gamma_1}) \mid \gamma_1 \in V_1 \wedge {\gamma_2}^{\gamma_1} \in V_4 \wedge \gamma_1 \sqsubseteq \gamma_2\}
\end{aligned}
$$

$$weight(e) = \begin{cases} 2 \text{ if } e \in E_2 \\ 1 \text{ if } e \in E_3 \cup E_4 \end{cases} \qquad \forall e \in E$$

Implementation 1. [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 $|\beta_1| + |\mathtt{Def}(\beta_2)|$.

We now prove the correctness of the implementation.

Lemma 4.2. 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 $|\beta_1| + |\mathtt{Def}(\beta_2)|$.

Sketch of Proof: ($\Rightarrow$) Let $\beta_1 \sqsubseteq \beta_2$. By Definition 8, there exists an ordering function f of β_1 to β_2. Consider the set given by $M = M_2 \cup M_3 \cup M_4$ where,

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

The proof consists of showing that M is a matching of the appropriate weight.

($\Leftarrow$) 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

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

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 $f(\gamma_1) = \mathit{undefined}$ otherwise. The proof consists of showing that f is a total function from β_1 to β_2, and that f is an ordering function of β_1 to β_2. $\square$

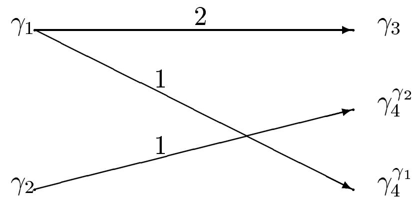
Theorem 4.3. The implementation of ordering satisfies its definition.

Proof. This follows from Lemma 4.2 and the fact that there cannot exist a matching of cost greater than $|\beta_1| + |\mathtt{Def}(\beta_2)|$ when testing $\beta_1 \sqsubseteq \beta_2$.

Example 4.6. Consider the abstract stores $\beta_1 = \{\gamma_1, \gamma_2\}$ and $\beta_2 = \{\gamma_3, \gamma_4\}$, where

$$\gamma_1 = \langle \oplus = \oplus\mathtt{P} + \top\mathtt{R}, \mathtt{One} \rangle \qquad \gamma_2 = \langle \oplus = \oplus\mathtt{P} + \ominus\mathtt{R}, \mathtt{Any} \rangle$$
$$\gamma_3 = \langle \oplus = \top\mathtt{P} + \top\mathtt{R}, \mathtt{One} \rangle \qquad \gamma_4 = \langle \oplus = \oplus\mathtt{P} + \top\mathtt{R}, \mathtt{Any} \rangle.$$

The matching graph G of β_1 to β_2 is given below:



We have that $\beta_1 \sqsubseteq \beta_2$ because the maximum matching of G has weight 3.

The following result gives the complexity of the ordering implementation.

Theorem 4.4. 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 [13] 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.

We now turn to the basic operation of CLP languages: *adding a constraint to a constraint store.* We make the operation slightly more general than needed to simplify the rest of the section.

Specification 4. [Adding an Abstract Constraint with Multiplicity] Operation $\uplus$: $AS_D \times AM_D \rightarrow AS_D$ should satisfy the following consistency condition. $\forall \theta_1, \theta_2 \in CS_D, \forall \beta \in AS_D, \forall \gamma \in AM_D$:

$$\theta_1 \in Cc(\beta) \;\wedge\; \theta_2 \in Cc(\gamma) \Rightarrow \theta_1 \,\underline{\cup}\, \theta_2 \;\in\; Cc(\beta \;\uplus\; \gamma).$$

Definition 4.13. 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

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

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 of its consistency is non-trivial and indicates why it is convenient to consider multisets (and not sets) of constraints in the concrete semantics.

Example 4.7. Let $\beta = \{\langle \oplus = \oplus \mathtt{T}, \mathtt{One} \rangle, \langle 0 \leq \oplus \mathtt{P}, \mathtt{One} \rangle\}$. Adding the abstraction of $0 \leq 2\mathtt{R} + 3\mathtt{B}$, i.e. $\langle 0 \leq \oplus \mathtt{R} + \oplus \mathtt{B}, \mathtt{One} \rangle$, to β leads to

$$\beta' = \{\langle \oplus = \oplus \mathtt{T}, \mathtt{One} \rangle,\; \langle 0 \leq \oplus \mathtt{P}, \mathtt{One} \rangle,\; \langle 0 \leq \oplus \mathtt{R} + \oplus \mathtt{B}, \mathtt{One} \rangle\}.$$

Adding the abstraction of $0 \leq 3\mathtt{R} + 4\mathtt{B}$, i.e. $\langle 0 \leq \oplus \mathtt{R} + \oplus \mathtt{B}, \mathtt{One} \rangle$, to β' gives the store $\{\langle \oplus = \oplus \mathtt{T}, \mathtt{One} \rangle,\; \langle 0 \leq \oplus \mathtt{P}, \mathtt{One} \rangle,\; \langle 0 \leq \oplus \mathtt{R} + \oplus \mathtt{B}, \mathtt{One} \rangle,\; \langle 0 \leq \oplus \mathtt{R} + \oplus \mathtt{B}, \mathtt{Any} \rangle\}$.

Theorem 4.5. Let β be an abstract store and γ be an abstract constraint with multiplicity.

(i) If $\theta_1 \in Cc_i(\beta)$ and $\theta_2 \in Cc(\gamma)$, then $\theta_1 \,\underline{\cup}\, \theta_2 \in Cc_i(\beta \uplus \gamma)$.

(ii) If $\theta_1 \in Cc(\beta)$ and $\theta_2 \in Cc(\gamma)$, then $\theta_1 \,\underline{\cup}\, \theta_2 \in Cc(\beta \uplus \gamma)$.

SKETCH OF PROOF: Let $\gamma = \langle \sigma, \mu \rangle$. If $\theta_1 \in Cc_i(\beta)$, then there exists a lifting function f of θ_1 to β by Lemma 4.1. Consider the function $f' : \theta_1 \,\underline{\cup}\, \theta_2 \rightarrow \beta \uplus \gamma$ given by

$$f'(\lambda) = f(\lambda) \qquad \text{for all } \lambda \in \theta_1.$$

$$f'(\lambda) = \begin{cases} \gamma & \text{if } \gamma \notin \beta \\ \langle \sigma, \mathtt{Any} \rangle & \text{if } \gamma = \langle \sigma, \mu \rangle \in \beta. \end{cases} \text{ for all } \lambda \in \theta_2.$$

The proof consists of showing that f' is a lifting function of $\theta_1 \,\underline{\cup}\, \theta_2$ to $\beta \uplus \gamma$. □

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

Definition 4.14. 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 4.3. Let β and β' be abstract stores.
(i) If $\theta \in Cc_i(\beta)$ and $\theta' \in Cc_i(\beta')$ then $\theta \sqcup \theta' \in Cc_i(\beta \biguplus \beta')$.
(ii) If $\theta \in Cc(\beta)$ and $\theta' \in Cc(\beta')$ then $\theta \sqcup \theta' \in Cc(\beta \biguplus \beta')$.

Example 4.8. The following example is based on the mortgage program `mg/4` of Example 2.1. We have that $\theta \in Cc_i(\beta)$ where $\theta = \{\ 0 < \mathtt{T},\ 0 \leq \mathtt{P},\ 0 = \mathtt{P} * 1.01 - \mathtt{R} - \mathtt{P1},\ 1 = \mathtt{T} - \mathtt{T1}\ \}$ and $\beta = \{\langle 0 < \oplus\mathtt{T}, \mathtt{One}\rangle,\ \langle 0 \leq \oplus\mathtt{P}, \mathtt{One}\rangle,\ \langle 0 = \oplus\mathtt{P} + \ominus\mathtt{R} + \ominus\mathtt{P1}, \mathtt{One}\rangle,\ \langle \oplus = \oplus\mathtt{T} + \ominus\mathtt{T1}, \mathtt{One}\rangle\}$. Also $\theta' \in Cc_i(\beta')$ where $\theta' = \{\ 0 = \mathtt{T1},\ 0 = \mathtt{B} - \mathtt{P1}\ \}$ and $\beta' = \{\langle 0 = \oplus\mathtt{T1}, \mathtt{One}\rangle, \langle 0 = \oplus\mathtt{B} + \ominus\mathtt{P1}, \mathtt{One}\rangle\}$. This gives by Lemma 4.3 that $\theta \sqcup \theta' \in Cc(\beta \biguplus \beta')$.

We now turn to the *upper bound of constraint stores.* The operation `UNION:` $AS_D \times AS_D \to AS_D$ is specified as follows.

Specification 5. `UNION:` $AS_D \times AS_D \to AS_D$ should satisfy the following consistency condition. $\forall\theta \in CS_D, \forall\beta_1 \in AS_D, \forall\beta_2 \in AS_D$:

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

We first define the upper bound of signs and multiplicities.

Definition 4.15. [Upper Bound on Signs] The upper bound operation on signs $\sqcup$: $\mathtt{Sign} \times \mathtt{Sign} \to \mathtt{Sign}$ is defined as $s_1 \sqcup s_2 = s_1$ if $s_1 = s_2$, $\top$ otherwise

Definition 4.16. [Upper Bound on Multiplicities] The upper bound operation on multiplicities $\sqcup$: $\mathtt{Mult} \times \mathtt{Mult} \to \mathtt{Mult}$ 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 $\{\langle \oplus = \oplus x_1 + \oplus x_2, \mathtt{One}\rangle\}$ and β_2 is $\{\langle \oplus = \oplus x_1 + \ominus x_2, \mathtt{One}\rangle\}$, both $\beta_3 = \{\langle \oplus = \oplus x_1 + \oplus x_2, \mathtt{ZeroOrOne}\rangle, \langle \oplus = \oplus x_1 + \ominus x_2, \mathtt{ZeroOrOne}\rangle\}$ and $\beta_4 = \{\langle \oplus = \oplus x_1 + \top x_2, \mathtt{One}\rangle\}$ 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 upper bound 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. We use the notation $\gamma :: \beta$ to denote the set $\{\gamma\} \cup \beta$ with $\gamma \notin \beta$.

Implementation 2. [Upper Bound on Abstract Stores] An upper bound operation on abstract stores `UNION:` $AS_D \times AS_D \to AS_D$ is any operation obtained by applying in a nondeterministic way the following set of rules.

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

2 *i*] $\mathtt{UNION}(\langle \sigma, \mathtt{One}\rangle :: \beta_1', \beta_2) = \mathtt{UNION}(\beta_1', \beta_2)\ \uplus\ \langle \sigma, \mathtt{ZeroOrOne}\rangle$

$$
\begin{aligned}
ii]\ & \texttt{UNION}(\beta_1, \langle\sigma, \texttt{One}\rangle :: \beta_2') = \texttt{UNION}(\beta_1, \beta_2') \ \uplus \ \langle\sigma, \texttt{ZeroOrOne}\rangle \\
3\ i]\ & \texttt{UNION}(\langle\sigma, \mu\rangle :: \beta_1', \beta_2) = \texttt{UNION}(\beta_1', \beta_2) \ \uplus \ \langle\sigma, \mu\rangle \quad \text{if } \mu \neq \texttt{One} \\
ii]\ & \texttt{UNION}(\beta_1, \langle\sigma, \mu\rangle :: \beta_2') = \texttt{UNION}(\beta_1, \beta_2') \ \uplus \ \langle\sigma, \mu\rangle \quad \text{if } \mu \neq \texttt{One} \\
4]\ & \texttt{UNION}(\langle s_0\ \delta\ \sum_{i=1}^{n} s_i x_i, \mu\rangle :: \beta_1', \langle s_0'\ \delta\ \sum_{i=1}^{n} s_i' x_i, \mu'\rangle :: \beta_2') = \\
& \qquad \texttt{UNION}(\beta_1', \beta_2') \ \uplus \ \langle (s_0 \sqcup s_0')\ \delta\ \sum_{i=1}^{n} (s_i \sqcup s_i') x_i, \mu \sqcup \mu' \rangle
\end{aligned}
$$

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.

Example 4.9. Given the abstract stores $\{\langle 0 = \oplus\texttt{T}, \texttt{One}\rangle\}$ and $\{\langle \oplus = \oplus\texttt{T}, \texttt{One}\rangle,\ \langle 0 < \oplus\texttt{T}, \texttt{One}\rangle\}$, both $\{\langle 0 = \oplus\texttt{T}, \texttt{ZeroOrOne}\rangle,\ \langle \oplus = \oplus\texttt{T}, \texttt{ZeroOrOne}\rangle,\ \langle 0 < \oplus\texttt{T}, \texttt{ZeroOrOne}\rangle\}$ (applying rule 2 thrice) and $\{\langle \top = \oplus\texttt{T}, \texttt{One}\rangle,\ \langle 0 < \oplus\texttt{T}, \texttt{ZeroOrOne}\rangle\}$ (applying rule 4 and rule 2) are upper bounds.

The following theorem states that the implementation of the upper bound operation satisfies its specification.

Theorem 4.6. Let β_1 and β_2 be abstract stores and θ be a constraint store.
(i) $\theta \in Cc_i(\beta_1) \vee \theta \in Cc_i(\beta_2) \ \Rightarrow\ \theta \in Cc_i(\texttt{UNION}(\beta_1, \beta_2))$.
(ii) $\theta \in Cc(\beta_1) \vee \theta \in Cc(\beta_2) \ \Rightarrow\ \theta \in Cc(\texttt{UNION}(\beta_1, \beta_2))$.

We now turn to the *projection of a contraint store.* Figure 4.1 describes a simple projection algorithm based on the traditional Gaussian and Fourier eliminations which are standard in this area. Figure 4.2 presents the abstract algorithm. The algorithms are close to those in [28] but they are simplified thanks to the introduction of operations `Csplit` and `Asplit` which avoids much of the tedious case analysis. The algorithms are 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 uses the equation or its negation to eliminate x_v from each of the other constraints in the store. The elimination is achieved by applying `Celiminate` 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 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_top`, which is used in the abstract version in order to simplify the handling of $\top$ coefficients in Fourier elimination. Before eliminating variable x_v through Fourier elimination, `Asplit_top` is used to change the $\top$ coefficients of x_v in the store to $0, \oplus$ or $\ominus$.

```
Cproject_set(θ, V) {
if V = ∅ return θ;
else let V = {v} ∪ V' ∧ v ∉ V' in
    return Cproject_set(Cproject(θ, v), V');
}

Cproject(θ, v) {
if θ = ∅ return ∅;
else let θ = {λ} ∪ θ' in
    case op(λ) is '=' ∧ λ[v] > 0 :
        return Cgauss(θ',λ,v);
    case op(λ) is '=' ∧ λ[v] < 0 :
        return Cgauss(θ',Cneg(λ),v);
    otherwise:
        return Cfourier(θ, v);
}

Cgauss(θ,λ, v) {
if θ = ∅ return ∅;
else let θ = {λ'} ∪ θ' in
    return Cgauss(θ',λ,v) ∪
            {Celiminate(λ',λ,v)};
}

Cfourier(θ, v) {
if θ = ∅ return ∅;
else let θ = {λ}∪θ' in
    ⟨θ1^0, θ1^+, θ1^-⟩ := Csplit({λ}, v);
    ⟨θ2^0, θ2^+, θ2^-⟩ := Csplit(θ', v);
    return θ1^0 ∪ Cfourier_step(θ1^+,θ2^-,v)
                ∪ Cfourier_step(θ2^+,θ1^-,v)
                ∪ Cfourier(θ',v);
}

Cfourier_step(θ^+,θ^-,v) {
θ := ∅;
for each λ^+ ∈ θ^+ and λ^- ∈ θ^-
    θ := θ ∪ {Ccombine(λ^-,λ^+,v)};
return θ;
}

Csplit(θ, v) {
return Csplit_basic(θ, v)
}
```

```
Cneg(λ) {
for i = 0, ..., n
    λ'[i] := - λ[i];
op(λ') := '=';
return λ';
}

Celiminate(λ1,λ=^±,v) {
for i := 0, ..., n
    λ[i] := λ=^±[v] × λ1[i] − λ1[v] × λ=^±[i];
λ[v] := 0;
op(λ) := op(λ1);
return λ;
}

Ccombine(λ^-,λ^+,v) {
for i := 0, ..., n
    λ[i] := λ^+[v] × λ^-[i] − λ^-[v] × λ^+[i];
λ[v] := 0;
op(λ) := op(λ^-) ⋈ op(λ^+);
return λ;
}
```

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

```
Csplit_basic(θ, v) {
if θ = ∅ return ⟨∅, ∅, ∅⟩;
else let  θ = {λ} ∪ θ' in
    ⟨θ^0, θ^+, θ^-⟩ := Csplit_basic(θ', v);
    case λ[v] = 0:
        return ⟨θ^0 ∪ {λ}, θ^+, θ^-⟩;
    case op(λ) is '=' ∧ λ[v] > 0:
        return ⟨θ^0, θ^+ ∪ {λ}, θ^- ∪ {Cneg(λ)}⟩;
    case op(λ) is '=' ∧ λ[v] < 0:
        return ⟨θ^0, θ^+ ∪ {Cneg(λ)}, θ^- ∪ {λ}⟩;
    case op(λ) is not '=' ∧ λ[v] > 0:
        return ⟨θ^0, θ^+ ∪ {λ}, θ^-⟩;
    case op(λ) is not '=' ∧ λ[v] < 0:
        return ⟨θ^0, θ^+, θ^- ∪ {λ}⟩;
}
```

FIGURE 4.1. Concrete Projection Algorithms

```
Aproject_set(β, V) {
if V = ∅ return β;
else let V = {v} ∪ V′ ∧ v ∉ V′ in
    return Aproject_set(Aproject(β, v), V′);
}

Aproject(β, v) {
if β = ∅ return ∅;
else let β = {γ} ∪ β′ ∧ γ = ⟨σ, μ⟩ ∉ β′ in
    case μ = One ∧ op(σ) is '=' ∧ σ[v] = ⊕:
        return Agauss(β′,σ,v);
    case μ = One ∧ op(σ) is '=' ∧ σ[v] = ⊖:
        return Agauss(β′,Aneg(σ),v);
    otherwise:
        return Afourier(β,v);
}

Agauss(β,σ,v) {
if β = ∅ return ∅;
else let β = {γ′} ∪ β′, γ′ = ⟨σ′, μ′⟩ ∉ β′ in
    return Agauss(β′,σ,v) ⊎
            ⟨Aeliminate(σ′, σ,v), μ′⟩;
}

Afourier(β,v) {
if β = ∅ return ∅;
else let β = {γ′} ∪ β′, γ′ = ⟨σ′, μ′⟩ ∉ β′ in
    ⟨β1^0, β1^+, β1^-⟩ := Asplit({γ′},v);
    ⟨β2^0, β2^+, β2^-⟩ := Asplit(β′,v);
    if μ′ = Any then return
        β1^0 ⨄ Afourier_step(β1^+,β2^- ⨄ β1^-,v)
             ⨄ Afourier_step(β2^+ ⨄ β1^+,β1^-,v)
             ⨄ Afourier(β′,v);
    else return
        β1^0 ⨄ Afourier_step(β1^+,β2^-,v)
             ⨄ Afourier_step(β2^+,β1^-,v)
             ⨄ Afourier(β′,v);
}

Afourier_step(β^+,β^-,v) {
β := ∅;
for each ⟨σ^+, μ^+⟩ ∈ β^+ and ⟨σ^-, μ^-⟩ ∈ β^-
    β := β ⊎ ⟨Acombine(σ^-, σ^+,v), μ^- ⊔ μ^+⟩;
return β;
}

Asplit(β,v) {
return Asplit_basic(Asplit_top(β,v),v);
}

Aneg(σ) {
for i = 0, ..., n
    σ′[i] := - σ[i];
op(σ′) := '=';   return σ′;
}

Aeliminate(σ1,σ=^±,v) {
for i := 0, ..., n
    σ[i] := σ=^±[v] × σ1[i] − σ1[v] × σ=^±[i];
σ[v] := 0;   op(σ) := op(σ1);
return σ;
}

Acombine(σ^-,σ^+,v) {
for i := 0, ..., n
    σ[i] := σ^+[v] × σ^-[i] − σ^-[v] × σ^+[i];
σ[v] := 0;   op(σ) := op(σ^-) ⋈ op(σ^+);
return σ;
}

Asplit_top(β,v) {
if β = ∅ return ∅;
else let  β = {γ} ∪ β′ ∧ γ = ⟨σ, μ⟩ ∉ β′
          γ^0 = ⟨σ^0, μ ⊔ ZeroOrOne⟩
          γ^+ = ⟨σ^+, μ ⊔ ZeroOrOne⟩
          γ^- = ⟨σ^-, μ ⊔ ZeroOrOne⟩
          σ^0 = σ except that σ^0[v] = 0
          σ^+ = σ except that σ^+[v] = ⊕
          σ^- = σ except that σ^-[v] = ⊖ in
    case σ[v] = ⊤:
        return Asplit_top(β′,v) ⨄ {γ^0, γ^+, γ^-};
    otherwise:
        return Asplit_top(β′,v) ⊎ γ;
}

Asplit_basic(β,v) {
if β = ∅ return ⟨∅, ∅, ∅⟩;
else let  β = {γ} ∪ β′ ∧ γ = ⟨σ, μ⟩ ∉ β′ in
    ⟨β^0, β^+, β^-⟩ := Asplit_basic(β′,v);
    case σ[v] = 0:
        return ⟨β^0 ⊎ γ, β^+, β^-⟩;
    case op(σ) is '=' ∧ σ[v] = ⊕:
        return ⟨β^0, β^+ ⊎ γ, β^- ⊎ ⟨Aneg(σ), μ⟩⟩;
    case op(σ) is '=' ∧ σ[v] = ⊖:
        return ⟨β^0, β^+ ⊎ ⟨Aneg(σ), μ⟩, β^- ⊎ γ⟩;
    case op(σ) is not '=' ∧ σ[v] = ⊕:
        return ⟨β^0, β^+ ⊎ γ, β^-⟩;
    case op(σ) is not '=' ∧ σ[v] = ⊖:
        return ⟨β^0, β^+, β^- ⊎ γ⟩;
}
```

FIGURE 4.2. Abstract Projection Algorithms

CSplit_basic and ASplit_basic split a (concrete or abstract) store into three partitions with the coefficient of x_v positive, negative, or zero. Operation Csplit partitions the store into three sets depending upon the coefficient of x_v. Its abstract version needs to deal with the case where the coefficient of the abstract constraint is $\top$. This is handled in Asplit_top whereby, Asplit_basic mimics Csplit_basic almost line by line. 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 [28]. This is achieved by testing the multiplicity of the constraint.

Of course, in the abstract algorithm, operations and relations on signs replace operations and relations on coefficients. We need to assume operations like $+, -, \times$ on signs; these operations must be consistent approximations of the corresponding operations on $\Re$. For instance, $c_1 \in Cc(s_1) \wedge c_2 \in Cc(s_2) \Rightarrow c_1 + c_2 \in Cc(s_1 + s_2)$. Operations Aneg, Aeliminate, and Acombine mimic Cneg, Celiminate, and Ccombine respectively, by performing operations on signs instead of on coefficients. Operation Agauss mimics operation Cgauss line by line. Contrary to the algorithm in [28], the abstract Gaussian elimination algorithm does not split the $\top$ coefficients of the variable being eliminated into 0, $\oplus$ and $\ominus$. This enables it to be more precise in some cases.

Example 4.10. Let $\gamma_1 = \langle 0 \leq \oplus\mathtt{R} + \oplus\mathtt{B}, \mathtt{One}\rangle$, $\gamma_2 = \langle 0 = \oplus\mathtt{P} + \top\mathtt{R} + \ominus\mathtt{B}, \mathtt{One}\rangle$ and $\beta = \{\gamma_1, \gamma_2\}$. Then

$$\mathtt{Asplit}(\{\gamma_1\},\mathtt{R}) = \langle \beta_1^0, \beta_1^+, \beta_1^- \rangle = \langle \emptyset, \{\langle 0 \leq \oplus\mathtt{R} + \oplus\mathtt{B}, \mathtt{One}\rangle\}, \emptyset \rangle$$

$$\begin{aligned}\mathtt{Asplit}(\{\gamma_2\},\mathtt{R}) = \langle \beta_2^0, \beta_2^+, \beta_2^- \rangle = \langle\ & \{\langle 0 = \oplus\mathtt{P} + \ominus\mathtt{B}, \mathtt{ZeroOrOne}\rangle\},\\ & \{\langle 0 = \oplus\mathtt{P} + \oplus\mathtt{R} + \ominus\mathtt{B}, \mathtt{ZeroOrOne}\rangle\},\\ & \{\langle 0 = \oplus\mathtt{P} + \ominus\mathtt{R} + \ominus\mathtt{B}, \mathtt{ZeroOrOne}\rangle\}\rangle\end{aligned}$$

According to the algorithm,

$$\begin{aligned}\mathtt{Afourier}(\beta,\mathtt{R}) &= \beta_1^0 \uplus \mathtt{Afourier\_step}(\beta_1^+, \beta_2^-, \mathtt{R}) \uplus\\ &\quad\ \mathtt{Afourier\_step}(\beta_2^+, \beta_1^-, \mathtt{R}) \uplus \mathtt{Afourier}(\{\gamma_2\},\mathtt{R})\\ &= \emptyset \uplus \{\langle 0 \leq \oplus\mathtt{P} + \top\mathtt{B}, \mathtt{ZeroOrOne}\rangle\} \uplus\\ &\quad\ \emptyset \uplus \mathtt{Afourier}(\{\gamma_2\},\mathtt{R})\end{aligned}$$

By similar reasoning, we have that

$$\mathtt{Afourier}(\{\gamma_2\},\mathtt{R}) = \{\langle 0 = \oplus\mathtt{P} + \ominus\mathtt{B}, \mathtt{ZeroOrOne}\rangle\}$$

This gives

$$\mathtt{Afourier}(\beta,\mathtt{R}) = \{\langle 0 \leq \oplus\mathtt{P} + \top\mathtt{B}, \mathtt{ZeroOrOne}\rangle, \langle 0 = \oplus\mathtt{P} + \ominus\mathtt{B}, \mathtt{ZeroOrOne}\rangle\}$$

Finally, Cproject_set and Aproject_set merely extend the projection operation to project a set of variables from a store, rather than just one variable. The correctness proofs of the operations above can be found in [36].

Theorem 4.7 Project. Let $v \in \mathcal{I}$, θ be a store and β be an abstract store.

$$\theta \in Cc(\beta) \ \Rightarrow\ \mathtt{Cproject}(\theta, v) \in Cc(\mathtt{Aproject}(\beta, v)).$$

Corollary 4.1 Project Set. Let $V \in 2^{\mathcal{I}}$, θ be a store and β be an abstract store.

$$\theta \in Cc(\beta) \;\Rightarrow\; \texttt{Cproject\_set}(\theta, V) \in Cc(\texttt{Aproject\_set}(\beta, V)).$$

Discussion. It is interesting to discuss some of our improvements over [28]. The first improvement is on the accuracy of abstract Fourier elimination. In [28], when Fourier eliminating x from a constraint that contains x with coefficient $\top$, that constraint is always combined with itself (as it may represent two concrete constraints with opposite coefficients for x). This is avoided in our domain by explicitly checking the multiplicity of the constraint and combining it with itself only if its multiplicity is Any. For example, projecting x from $\{\langle \top = \top x + \oplus y, \texttt{ZeroOrOne}\rangle\}$ leads to the store $\{\langle \top = \oplus y, \texttt{ZeroOrOne}\rangle, \langle \top = \oplus y, \texttt{Any}\rangle\}$ using the algorithm of [28]. The first abstract constraint in this store comes from taking the $\top$ coefficient of x to be 0, while the second constraint comes from taking the $\top$ coefficient of x to be $\oplus$ and $\ominus$ and combining the two. This store cannot be deduced to be definitely satisfiable by projecting y as it potentially contains more than one different assignment of a value to y. Our improved algorithm would lead to the store $\{\langle \top = \oplus y, \texttt{ZeroOrOne}\rangle\}$, which can be easily seen to be definitely satisfiable.

The second improvement is on the accuracy of abstract Gauss elimination. Consider the following example. Using the algorithms of [28], projecting x from $\{\langle \oplus = \oplus x, \texttt{One}\rangle, \langle \top = \top x + \oplus y, \texttt{One}\rangle\}$ involves considering the cases 0, $\oplus$ and $\ominus$ for the $\top$ coefficient of x in the second constraint, leading to $\{\langle \top = \oplus y, \texttt{ZeroOrOne}\rangle, \langle \top = \oplus y, \texttt{Any}\rangle\}$ which may or may not be satisfiable. However, by directly substituting $\langle \oplus = \oplus x, \texttt{One}\rangle$ into the second constraint, we get a simpler abstract store $\{\langle \top = \oplus y, \texttt{One}\rangle\}$ that accurately describes the result of projecting x and which can be deduced to be definitely satisfiable. Our abstract Gauss elimination algorithm produces the above store by avoiding the imprecise splitting of the coefficient $\top$.

4.3. Satisfiability

Reordering algorithms for CLP programs [36] are based on one fundamental operation: testing if a constraint store $\theta_1 \cup \theta_2$ is satisfiable whenever θ_1 is satisfiable. This operation makes it possible to determine if a constraint can prune the search space in a goal. This section discusses how to use LSign for abstracting this operation.

Consider first the problem of *determining whether a constraint store is satisfiable.* In the concrete domain, this problem can be formalized by a function Cis_sat : $CS_D \rightarrow$ Boolean which takes a constraint store θ and returns a Boolean value which is true if θ is satisfiable and false otherwise. An implementation of Cis_sat is given in Figure 4.3. It consists of the well known technique of projecting all the variables from the store and checking if the resulting ground constraints (i.e. constraints with zero coefficients for all variables) are all trivially satisfiable. Its abstract counterpart Ais_sat, also given in Figure 4.3, makes a conservative approximation to Cis_sat.

Theorem 4.8 Is Satisfiable. Let θ be a store and β be an abstract store. $\theta \in Cc(\beta) \Rightarrow (\texttt{Ais\_sat}(\beta) \Rightarrow \texttt{Cis\_sat}(\theta))$.

```
Cis_sat(θ) {
θ_p = Cproject_set(θ, I);
return Cis_triv_sat(θ_p);
}

Cis_triv_sat(θ) {
if θ = ∅ return true ;
else let θ = {λ} ⊍ θ' in
    case λ is c_0 < Σ_{i=1}^n 0x_i ∧ c_0 < 0:
    case λ is c_0 = Σ_{i=1}^h 0x_i ∧ c_0 = 0:
    case λ is c_0 ≤ Σ_{i=1}^h 0x_i ∧ c_0 ≤ 0:
        return Cis_triv_sat(θ');
    otherwise:
        return false;
}
```

```
Ais_sat(β) {
β_p = Aproject_set(β, I);
return Ais_triv_sat(β_p);
}

Ais_triv_sat(β) {
if β = ∅ return true ;
else let β = {γ} ∪ β' ∧ γ = ⟨σ, μ⟩ ∉ β' in
    case σ is s_0 < Σ_{i=1}^n 0x_i ∧ s_0 = ⊖:
    case σ is s_0 = Σ_{i=1}^h 0x_i ∧ s_0 = 0:
    case σ is s_0 ≤ Σ_{i=1}^h 0x_i ∧ (s_0 = 0 ∨ s_0 = ⊖):
        return Ais_triv_sat(β');
    otherwise:
        return false;
}
```

FIGURE 4.3. Satisfiability Algorithms

Example 4.11. Projecting all the variables from $\{0 \leq \mathtt{R}+\mathtt{B},\ 0 = \mathtt{B},\ 3 = \mathtt{R}\}$ gives the constraint store $\{-3 \leq 0\}$, indicating that the original constraint store is satisfiable. Projecting all the variables from $\{\langle\ 0 \leq \oplus\mathtt{R} + \oplus\mathtt{B},\ \mathtt{One}\ \rangle,\ \langle\ 0 = \oplus\mathtt{B},\ \mathtt{One}\ \rangle,\ \langle\ \oplus = \oplus\mathtt{R},\ \mathtt{One}\ \rangle\}$ gives the abstract store $\{\langle\ \ominus \leq 0,\ \mathtt{One}\ \rangle\}$, indicating that all the constraint stores in its concretization are satisfiable.
Projecting all the variables from $\{\langle\ \top \leq \oplus\mathtt{R} + \oplus\mathtt{B},\ \mathtt{One}\ \rangle,\ \langle\ 0 = \oplus\mathtt{B},\ \mathtt{One}\ \rangle,\ \langle\ \oplus = \oplus\mathtt{R},\ \mathtt{One}\ \rangle\}$ gives the abstract store $\{\langle\ \top \leq 0,\ \mathtt{One}\ \rangle\}$, indicating that some of the constraint stores in its concretization *may not* be satisfiable.

Consider now the problem of *determining whether a constraint store is unsatisfiable.* In the concrete domain, this problem can be formalized by the function $\mathtt{Cis\_unsat} : CS_D \rightarrow$ Boolean which takes a constraint store θ and returns a Boolean value which is true if θ is unsatisfiable and false otherwise. The function `Cis_unsat` is exactly the logical negation of the function `Cis_sat`. The definition of `Cis_unsat` (Figure 4.4) is defined by projecting all the variables from the store and checking if any of the resulting ground constraints (i.e. constraints with zero coefficients for all variables) is trivially unsatisfiable. Its abstract counterpart `Ais_unsat`, also given in Figure 4.4, makes a conservative approximation to `Cis_unsat`.

Theorem 4.9 Is Unsatisfiable. Let θ be a store and β be an abstract store. $\theta \in Cc(\beta) \Rightarrow (\mathtt{Ais\_unsat}(\beta) \Rightarrow \mathtt{Cis\_unsat}(\theta))$.

Note that `Ais_unsat` is not the logical negation of `Ais_sat`, unlike in the concrete domain.

Example 4.12. Projecting all the variables from $\{0 \leq \mathtt{R} + \mathtt{B},\ 0 = \mathtt{B},\ -3 = \mathtt{R}\}$

```
Cis_unsat(θ) {
θ_p = Cproject_set(θ, I);
return Cis_triv_unsat(θ_p);
}

Cis_triv_unsat(θ) {
if θ = ∅ return true ;
else let θ = {λ} ⊍ θ' in
    case λ is c_0 < Σ_{i=1}^n 0x_i ∧ c_0 ≥ 0:
    case λ is c_0 = Σ_{i=1}^h 0x_i ∧ c_0 ≠ 0:
    case λ is c_0 ≤ Σ_{i=1}^h 0x_i ∧ c_0 > 0:
        return true;
    otherwise:
        return Cis_triv_unsat(θ');
}

Ais_unsat(β) {
β_p = Aproject_set(β, I);
return Ais_triv_unsat(β_p);
}

Ais_triv_unsat(β) {
if β = ∅ return true ;
else let β = {γ} ∪ β' ∧ γ = ⟨σ, μ⟩ ∉ β' in
    case σ is s_0 < Σ_{i=1}^n 0x_i ∧ (s_0 = 0 ∨ s_0 = ⊕)
            ∧ μ = One:
    case σ is s_0 = Σ_{i=1}^n 0x_i ∧ (s_0 = ⊕ ∨ s_0 = ⊖)
            ∧ μ = One:
    case σ is s_0 ≤ Σ_{i=1}^n 0x_i ∧ s_0 = ⊕
            ∧ μ = One:
        return true;
    otherwise:
        return Ais_triv_unsat(β');
}
```

FIGURE 4.4. Unsatisfiability Algorithms

gives the constraint store $\{3 \leq 0\}$, indicating that the original constraint store is unsatisfiable.
Projecting all the variables from $\{\langle\ 0 \leq \oplus\mathtt{R} + \oplus\mathtt{B},\ \mathtt{One}\ \rangle,\ \langle\ 0 = \oplus\mathtt{B},\ \mathtt{One}\ \rangle,\ \langle\ \ominus = \oplus\mathtt{R},\ \mathtt{One}\ \rangle\}$ gives the abstract store $\{\langle\ \oplus\ \leq\ 0,\ \mathtt{One}\ \rangle\}$, indicating that all the constraint stores in its concretization are unsatisfiable.
Projecting all the variables from $\{\langle\ \top \leq \oplus\mathtt{R} + \oplus\mathtt{B},\ \mathtt{One}\ \rangle,\ \langle\ 0 = \oplus\mathtt{B},\ \mathtt{One}\ \rangle,\ \langle\ \oplus = \oplus\mathtt{R},\ \mathtt{One}\ \rangle\}$ gives the abstract store $\{\langle\ \top \leq 0,\ \mathtt{One}\ \rangle\}$, indicating that some of the constraint stores in its concretization *may not* be unsatisfiable.

Finally, consider the problem of *conditional satisfiability of constraint stores.* In the concrete domain, this problem can be formalized by the function Cis_cond_sat : $CS_D \times CS_D \rightarrow$ Boolean which takes two constraint stores θ_1 and θ_2 and returns true if the conjunction $\theta_1 \uplus \theta_2$ is satisfiable whenever θ_1 is satisfiable. In other words,

$$\begin{aligned}\mathtt{Cis\_cond\_sat}(\theta_1,\theta_2) &= \mathtt{Cis\_sat}(\theta_1) \Rightarrow \mathtt{Cis\_sat}(\theta_1 \uplus \theta_2)\\ &= \mathtt{Cis\_unsat}(\theta_1)\ \vee\ \mathtt{Cis\_sat}(\theta_1 \uplus \theta_2).\end{aligned}$$

The need for such an operation arises when it is necessary to verify that adding any constraint store in the concretization of β_2 to any satisfiable constraint store in the concretization of β_1 does not cause the resulting constraint store to become unsatisfiable. The operation is the cornerstone of the analysis for reordering constraints in CLP($\Re_{Lin}$) programs. While the definition of Cis_cond_sat is straightforward, it does not lend itself to a straightforward satisfactory abstraction. A naive implementation of Ais_cond_sat is

$$\mathtt{Ais\_cond\_sat}(\beta_1,\beta_2) = \mathtt{Ais\_unsat}(\beta_1)\ \vee\ \mathtt{Ais\_sat}(\beta_1 \biguplus \beta_2).$$

However, this naive implementation is not sufficiently accurate in practice because the operations Ais_sat and Ais_unsat are not logical negations of one another. It becomes necessary to transform the store β_1 to remove sources of unsatisfiability in β_1 before checking the satisfiability of $\beta_1 \uplus \beta_2$. More formally, we are looking forward an implementation like

$$\texttt{Ais\_cond\_sat}(\beta_1, \beta_2) = \texttt{Ais\_unsat}(\beta_1) \ \vee \ \texttt{Ais\_sat}(\beta' \biguplus \beta_2).$$

where β' is an abstract description of $\{\theta \in Cc(\beta_1) \mid \texttt{Cis\_sat}(\theta)\}$ as accurate as possible. At this aim, we introduce four transformations to remove possible sources of unsatisfiability. Two of these, i.e., eliminating with equations and projecting irrelevant variables, serve to make the information available on the constraint store more explicit. The third, i.e., reducing top coefficients, corresponds to making the signs of the abstract store more accurate. It is to be performed before eliminating with equations so that the sign information in inequalities is not lost. The final transformation, i.e., removing ground constraints is to be applied last as the previous transformations may introduce ground constraints.

Removing Ground Constraints. In the concrete domain, the transformation Cred_gnd (Figure 4.5) removes all ground constraints (i.e. constraints with zero coefficients for all variables) from a constraint store, and it is an equivalence transformation for satisfiable constraint stores.

Proposition 4.1. Let θ be a constraint store. Then θ satisfiable $\Rightarrow (\theta \leftrightarrow \texttt{Cred\_gnd}(\theta))$.

While Cred_gnd is not a very useful transformation for constraint stores, the following example indicates why its abstraction Ared_gnd (given in Fig. 4.5) is important to improve the accuracy of conditional satisfiability in the abstract domain.

Example 4.13. Let $\beta_1 = \{\langle \top = \oplus\texttt{X}, \texttt{One}\rangle, \langle \top < 0, \texttt{One}\rangle\}$ and $\beta_2 = \{\langle 0 = \oplus\texttt{X} + \oplus\texttt{Y}, \texttt{One}\rangle$. Because β_2 only represents stores that have a simple linear relation between X and Y, and β_1 represents stores that do not involve Y at all, it can be seen that for any satisfiable constraint store $\theta_1 \in Cc(\beta_1)$ and constraint store $\theta_2 \in Cc(\beta_2)$, we have that $\texttt{Cis\_sat}(\theta_1 \mathbin{\underline{\cup}} \theta_2)$ is true. We expect therefore that $\texttt{Ais\_cond\_sat}(\beta_1, \beta_2)$ returns true. However we see that

$$\begin{aligned}\texttt{Ais\_cond\_sat}(\beta_1, \beta_2) &= \texttt{Ais\_unsat}(\beta_1) \ \vee \ \texttt{Ais\_sat}(\beta_1 \uplus \beta_2) \\ &= \text{false} \ \vee \ \texttt{Ais\_triv\_sat}(\{\langle \top < 0, \texttt{One}\rangle\}) \\ &= \text{false}.\end{aligned}$$

This inaccuracy arises because of the possibly unsatisfiable ground constraint $\langle \top < 0, \texttt{One}\rangle$ in β_1. However, there is another abstract store $\beta_1' = \{\langle \top = \oplus X, \texttt{One}\rangle\}$ which includes all the satisfiable constraint stores of $Cc(\beta_1)$ in its concretization and such that $\texttt{Ais\_sat}(\beta_1' \uplus \beta_2)$ is true. Moreover, $\beta_1' = \texttt{Ared\_gnd}(\beta_1)$.

An improved implementation of Ais_cond_sat, may therefore be given by

$$\texttt{Ais\_cond\_sat}(\beta_1, \beta_2) = \texttt{Ais\_unsat}(\beta_1) \ \vee \ \texttt{Ais\_sat}(\texttt{Ared\_gnd}(\beta_1) \biguplus \beta_2).$$

Eliminating Equations. In the concrete domain, the transformation Cred_eqn (Figure 4.5) looks for an equation for each variable and, if possible, uses the equation to eliminate the variable from the rest of the store (operation `Cred_eqn_step`). This corresponds to substituting the value of the variable in the rest of the store, but not removing the equation used to perform the substitution. `Cred_eqn` is an equivalence transformation on stores, i.e., the following proposition holds.

Proposition 4.2. Let $v \in \mathcal{I}, V \in 2^{\mathcal{I}}$ *and* θ *be a store. Then*

$$\theta \leftrightarrow \texttt{Cred\_eqn\_step}(\theta, v) \leftrightarrow \texttt{Cred\_eqn\_set}(\theta, V) \leftrightarrow \texttt{Cred\_eqn}(\theta).$$

The following example indicates why the abstraction of `Cred_eqn`, i.e. `Ared_eqn` (given in Figure 4.5) is important for the accuracy of conditional satisfiability.

Example 4.14. Let $\beta_1 = \{\langle \oplus = \oplus X, \texttt{One}\rangle, \langle \top < \oplus X, \texttt{One}\rangle\}$ and $\beta_2 = \{\langle 0 = \oplus X + \oplus Y, \texttt{One}\rangle\}$. For any satisfiable constraint stores $\theta_1 \in Cc(\beta_1)$ and $\theta_2 \in Cc(\beta_2)$, we have that $\texttt{Cis\_sat}(\theta_1 \uplus \theta_2)$ is true. However we see that

$$\begin{aligned}\texttt{Ais\_cond\_sat}(\beta_1, \beta_2) &= \texttt{Ais\_unsat}(\beta_1) \ \vee \ \texttt{Ais\_sat}(\beta_1 \biguplus \beta_2) \\ &= \text{false} \ \vee \ \texttt{Ais\_triv\_sat}(\{\langle \top < 0, \texttt{One}\rangle\}) \\ &= \text{false}.\end{aligned}$$

This inaccuracy arises because the two abstract constraints in β_1 may be potentially unsatisfiable together (combining to produce $\top < 0$). It can be observed however that there is another abstract store $\beta_1' = \{\langle \oplus = \oplus X, \texttt{One}\rangle\}$ which includes all the satisfiable constraint stores of $Cc(\beta_1)$ in its concretization (under equivalence closure) and such that $\texttt{Ais\_sat}(\beta_1' \biguplus \beta_2)$ is true. Moreover, $\beta_1' = \texttt{Ared\_gnd}(\texttt{Ared\_eqn}(\beta_1))$.

It is important to first use the equations to simplify the store and then use `Ared_gnd` to remove ground constraints. This is because `Ared_eqn` may introduce ground constraints when it eliminates with equations. An improved implementation of `Ais_cond_sat`, may therefore be given as

$$\texttt{Ais\_cond\_sat}(\beta_1, \beta_2) = \texttt{Ais\_unsat}(\beta_1) \ \vee \ \texttt{Ais\_sat}(\texttt{Ared\_gnd}(\texttt{Ared\_eqn}(\beta_1)) \biguplus \beta_2).$$

Reducing Top Coefficients. The third transformation does not have any counterpart in the concrete domain. It consists of reducing the number of top coefficients in the abstract store β_1 so as to improve its accuracy. The following example motivates the transformation.

Example 4.15. Let $\beta_1 = \{\langle \top = \oplus X, \texttt{One}\rangle, \langle 0 < \oplus X, \texttt{One}\rangle\}$ and $\beta_2 = \{\langle 0 = \oplus X + \oplus Y, \texttt{One}\rangle\}$. Again it can be seen that for any satisfiable constraint store $\theta_1 \in Cc(\beta_1)$ and constraint store $\theta_2 \in Cc(\beta_2)$, we have that $\texttt{Cis\_sat}(\theta_1 \uplus \theta_2)$ is true. However we see that

$$\begin{aligned}\texttt{Ais\_cond\_sat}(\beta_1, \beta_2) &= \texttt{Ais\_unsat}(\beta_1) \ \vee \ \texttt{Ais\_sat}(\beta_1 \biguplus \beta_2) \\ &= \text{false} \ \vee \ \texttt{Ais\_triv\_sat}(\{\langle \top < 0, \texttt{One}\rangle\}) \\ &= \text{false}.\end{aligned}$$

This inaccuracy arises because of the top coefficient in β_1. It can be observed that there is another abstract store $\beta_1' = \{\langle \oplus = \oplus X, \texttt{One}\rangle, \langle 0 < \oplus X, \texttt{One}\rangle\}$ which includes all the satisfiable constraint stores of $Cc(\beta_1)$ in its concretization and such that $\texttt{Ais\_sat}(\beta_1' \biguplus \beta_2)$ is true.

The basic idea is to consider the cases 0, $\oplus$ and $\ominus$ for each $\top$ coefficient in the store β_1 and see if any two of them make the store unsatisfiable (using `Ais_unsat`). In that case, the $\top$ coefficient can be replaced by the third sign. To do this in general for a store would be a very expensive operation. We therefore present a more specific version of the transformation which captures most of the cases that occur in practice. The transformation `Ared_top` (Figure 4.5) uses any inequality that restricts the sign of a variable, to refine any equation that assigns $\top$ to that variable. It is important to perform `Ared_top` before performing `Ared_eqn`, in order that any sign restricting inequality be used to make an equation more accurate before eliminating with that equation.

Reduction Operation. The above transformations, aimed at removing unsatisfiable stores, can all be put together in a transformation called the definite satisfiability *reduction* (Figure 4.5). The definition of `Areduce` retains all the satisfiable stores in the concretization, as stated by the following theorem.

Theorem 4.10 Reduce. Let θ be a store and β be an abstract store. Then

$$\theta \in Cc(\beta) \ \wedge\ \theta \text{ satisfiable } \Rightarrow\ \theta \in Cc(\texttt{Areduce}(\beta))$$

Using `Areduce`, the operation `Ais_cond_sat` can be implemented in a more accurate fashion as follows:

$$\texttt{Ais\_cond\_sat}(\beta_1, \beta_2) = \texttt{Ais\_unsat}(\beta_1)\ \vee\ \texttt{Ais\_sat}(\texttt{Areduce}(\beta_1) \uplus \beta_2).$$

Projecting Irrelevant Variables. Even the above implementation of `Ais_cond_sat` is not sufficiently accurate, as can be seen by the following example.

Example 4.16. Let $\beta_1 = \{\langle \top < \top X, \texttt{One}\rangle\}$ and $\beta_2 = \{\langle 0 = \oplus X + \oplus Y, \texttt{One}\rangle\}$. It is obvious that for any satisfiable constraint store $\theta_1 \in Cc(\beta_1)$ and constraint store $\theta_2 \in Cc(\beta_2)$, we have that $\texttt{Cis\_sat}(\theta_1 \uplus \theta_2)$. We expect therefore that $\texttt{Ais\_cond\_sat}(\beta_1, \beta_2)$ returns true. However we see that

$$\begin{aligned}\texttt{Ais\_cond\_sat}(\beta_1, \beta_2) &= \texttt{Ais\_unsat}(\beta_1)\ \vee\ \texttt{Ais\_sat}(\texttt{Areduce}(\beta_1) \uplus \beta_2)\\ &= \text{false}\ \vee\ \texttt{Ais\_triv\_sat}(\{\langle \top < 0, \texttt{ZeroOrOne}\rangle\})\\ &= \text{false}\end{aligned}$$

Intuitively, the inaccuracy is caused because β_1 contains unsatisfiable stores in its concretization, however the transformations performed by `Areduce` are not able to remove these unsatisfiable stores.

To understand how to overcome this limitation it is instructive to look at the concrete domain. If $\theta_1 \in Cc(\beta_1)$ and $\theta_2 \in Cc(\beta_2)$,

$$\begin{aligned}&\texttt{Cis\_sat}(\theta_1 \uplus \theta_2)\\ &= \texttt{Cis\_triv\_sat}(\texttt{Cproject\_set}(\theta_1 \uplus \theta_2, \mathcal{I}))\\ &= \texttt{Cis\_triv\_sat}(\texttt{Cproject\_set}(\texttt{Cproject\_set}(\theta_1, \texttt{Var}(\theta_1) \setminus \texttt{Var}(\theta_2)) \uplus \theta_2, \mathcal{I}))\\ &= \texttt{Cis\_sat}(\texttt{Cproject\_set}(\theta_1, \texttt{Var}(\theta_1) \setminus \texttt{Var}(\theta_2)) \uplus \theta_2)\end{aligned}$$

This suggests our final implementation for conditional satisfiability.

$\texttt{Creduce}(\theta)$ {
return $\texttt{Cred\_gnd}(\texttt{Cred\_eqn}(\theta))$;
}

$\texttt{Cred\_gnd}(\theta)$ {
if $\theta = \emptyset$ **return** $\emptyset$;
else let $\theta \ = \ \{\lambda\} \uplus \theta'$ **in**
  case λ is $c_0 \ \delta \ \sum_{i=1}^{n} 0x_i$:
    return $\texttt{Cred\_gnd}(\theta')$;
  otherwise:
    return $\texttt{Cred\_gnd}(\theta') \uplus \{\lambda\}$;
}

$\texttt{Cred\_eqn}(\theta)$ {
return $\texttt{Cred\_eqn\_set}(\theta, \mathcal{I})$;
}

$\texttt{Cred\_eqn\_set}(\theta, V)$ {
if $V = \emptyset$ **return** θ;
else let $V \ = \{v\} \ \cup \ V' \ \wedge \ v \notin V'$ **in**
  $\theta_v := \texttt{Cred\_eqn\_step}(\theta, v)$;
  return $\texttt{Cred\_eqn\_set}(\theta_v, V')$;
}

$\texttt{Cred\_eqn\_step}(\theta, v)$ {
if $\theta = \emptyset$ **return** $\emptyset$;
else let $\theta \ = \ \{\lambda\} \uplus \theta'$ **in**
  case $\mathrm{op}(\lambda)$ is '=' $\wedge \ \lambda[v] > 0$:
    return $\texttt{Cgauss}(\theta', \lambda, v) \uplus \{\lambda\}$;
  case $\mathrm{op}(\lambda)$ is '=' $\wedge \ \lambda[v] < 0$:
    return $\texttt{Cgauss}(\theta', \texttt{Cneg}(\lambda), v) \uplus \{\lambda\}$;
  otherwise:
    return θ;
}

$\texttt{Areduce}(\beta)$ {
return $\texttt{Ared\_gnd}(\texttt{Ared\_eqn}(\texttt{Ared\_top}(\beta)))$;
}

$\texttt{Ared\_gnd}(\beta)$ {
if $\beta = \emptyset$ **return** $\emptyset$;
else let $\beta \ = \ \{\gamma\} \cup \beta' \ \wedge \ \gamma = \langle \sigma, \mu \rangle \notin \beta'$ **in**
  case σ is $s_0 \ \delta \ \sum_{i=1}^{n} 0x_i$:
    return $\texttt{Ared\_gnd}(\beta')$;
  otherwise:
    return $\texttt{Ared\_gnd}(\beta') \uplus \gamma$;
}

$\texttt{Ared\_eqn}(\beta)$ {
return $\texttt{Ared\_eqn\_set}(\beta, \mathcal{I})$;
}

$\texttt{Ared\_eqn\_set}(\beta, V)$ {
if $V = \emptyset$ **return** β;
else let $V \ = \{v\} \ \cup \ V' \ \wedge \ v \notin V'$ **in**
  $\beta_v := \texttt{Ared\_eqn\_step}(\beta, v)$;
  return $\texttt{Ared\_eqn\_set}(\beta_v, V')$;
}

$\texttt{Ared\_eqn\_step}(\beta, v)$ {
if $\beta = \emptyset$ **return** $\emptyset$;
else let $\beta \ = \ \{\gamma\} \cup \beta' \ \wedge \ \gamma = \langle \sigma, \mu \rangle \notin \beta'$ **in**
  case $\mu = \texttt{One} \wedge \mathrm{op}(\sigma)$ is '=' $\wedge \ \sigma[v] = \oplus$:
    return $\texttt{Agauss}(\beta', \sigma, v) \uplus \gamma$;
  case $\mu = \texttt{One} \wedge \mathrm{op}(\sigma)$ is '=' $\wedge \ \sigma[v] = \ominus$:
    return $\texttt{Agauss}(\beta', \texttt{Aneg}(\sigma), v) \uplus \gamma$;
  otherwise:
    return β;
}

$\texttt{Ared\_top}(\beta)$ {
if $\beta = \emptyset$ **return** $\emptyset$;
else if $|\beta| = 1$ **return** β;
else let $\beta \ = \ \{\gamma_1, \gamma_2\} \cup \beta' \ \wedge \ \gamma_1 = \langle \sigma_1, \mu_1 \rangle \notin \beta' \ \wedge \ \gamma_2 = \langle \sigma_2, \mu_2 \rangle \notin \beta'$ **in**
  case $\mu_1 = \texttt{One} \ \wedge \ (\sigma_1$ is $\oplus \leq \oplus x_v \ \vee \ \sigma_1$ is $0 < \oplus x_v \ \vee \ \sigma_1$ is $\oplus < \oplus x_v)$
    $\wedge(\sigma_2$ is $\top = \oplus x_v \ \vee \ \sigma_2$ is $\top = \ominus x_v)$:
    return $\texttt{Ared\_top}(\beta' \uplus \gamma_1) \uplus \langle \oplus = \oplus x_v, \mu_2 \rangle$;
  case $\mu_1 = \texttt{One} \ \wedge \ (\sigma_1$ is $\oplus \leq \ominus x_v \ \vee \ \sigma_1$ is $0 < \ominus x_v \ \vee \ \sigma_1$ is $\oplus < \ominus x_v)$
    $\wedge(\sigma_2$ is $\top = \oplus x_v \ \vee \ \sigma_2$ is $\top = \ominus x_v)$:
    return $\texttt{Ared\_top}(\beta' \uplus \gamma_1) \uplus \langle \ominus = \oplus x_v, \mu_2 \rangle$;
  otherwise:
    return β;
}

FIGURE 4.5. Reduction Algorithms

Definition 4.17. [Is Conditionally Satisfiable] The abstract conditional satisfiability operation $\texttt{Ais\_cond\_sat} : AS_D \times AS_D \to \text{Boolean}$ is given as

$$\begin{aligned}&\texttt{Ais\_cond\_sat}(\beta_1, \beta_2)\\ &= \texttt{Ais\_unsat}(\beta_1) \ \vee \ \texttt{Ais\_sat}(\texttt{Areduce}(\texttt{Aproject\_set}(\beta_1, \texttt{Var}(\beta_1) \setminus \texttt{Var}(\beta_2))) \uplus \beta_2).\end{aligned}$$

where $\texttt{Var}(\beta)$ is the set of variables of β that have a nonzero coefficient in β.

Note that `Areduce` is applied to β_1 after projecting the variables of β_1 that do not occur in β_2. This is because the projection may make explicit sources of inconsistency (eg. ground unsatisfiable constraints) that can then be removed by `Areduce`. The above implementation of conditional satisfiability satisfies its specification.

Theorem 4.11 Conditional Satisfiability. Let θ_1, θ_2 be stores and β_1, β_2 be abstract stores. Then

$$\theta_1 \in Cc(\beta_1) \ \wedge \ \theta_2 \in Cc(\beta_2) \Rightarrow (\texttt{Ais\_cond\_sat}(\beta_1, \beta_2) \Rightarrow \texttt{Cis\_cond\_sat}(\theta_1, \theta_2))$$

4.4. The Power Domain $2^{\texttt{LSign}}$

In practice, the `LSign` domain is not always sufficiently accurate to perform a practical analysis of CLP programs. In particular, the upper bound operation may lose too much information to be of practical use. It may therefore be necessary to move to the power domain $2^{\texttt{LSign}}$ in order to get the required accuracy. This is a fairly standard construction in abstract interpretation [12]. The technical details of this domain lifting can be found in [34]. Here, we show the computation of the goal independent (or online) output in the $2^{\texttt{LSign}}$ domain, for the mortgage program `mg/4` of Example 2.1, as it results by our prototype implementation.

Example 4.17. [Computation of $2^{\texttt{LSign}}$ Output Description for `mg/4`] The abstract substitution describing the output of the first clause is

$$\{\{\langle 0 = \oplus\texttt{T}, \texttt{One}\rangle, \langle 0 = \oplus\texttt{P} + \ominus\texttt{B}, \texttt{One}\rangle\}\}$$

The abstract substitution describing the constraint store just before the recursive call in the second clause is

$$\{\{\langle 0 < \oplus\texttt{T}, \texttt{One}\rangle, \langle 0 \leq \oplus\texttt{P}, \texttt{One}\rangle, \langle 0 = \oplus\texttt{P} + \ominus\texttt{R} + \ominus\texttt{P1}, \texttt{One}\rangle, \langle \oplus = \oplus\texttt{T} + \ominus\texttt{T1}, \texttt{One}\rangle\}\}$$

Extending this with the denormalized output of the recursive call (when the recursive call returns the previous output) gives the abstract substitution

$$\begin{aligned}&\{\{\langle 0 < \oplus\texttt{T}, \texttt{One}\rangle, \langle 0 \leq \oplus\texttt{P}, \texttt{One}\rangle, \langle 0 = \oplus\texttt{P} + \ominus\texttt{R} + \ominus\texttt{P1}, \texttt{One}\rangle,\\ &\quad \langle \oplus = \oplus\texttt{T} + \ominus\texttt{T1}, \texttt{One}\rangle, \langle 0 = \oplus\texttt{T1}, \texttt{One}\rangle, \langle 0 = \oplus\texttt{P1} + \ominus\texttt{B}, \texttt{One}\rangle\}\}\end{aligned}$$

Restricting this to the head variables gives the following abstract substitution as the output of the second clause

$$\{\{\langle 0 < \oplus\texttt{T}, \texttt{One}\rangle, \langle \oplus = \oplus\texttt{T}, \texttt{One}\rangle, \langle 0 \leq \oplus\texttt{P}, \texttt{One}\rangle, \langle 0 = \oplus\texttt{P} + \ominus\texttt{R} + \ominus\texttt{B}, \texttt{One}\rangle\}\}$$

The union of this with the previously computed output of the first clause [7] gives the following abstract substitution as the updated output of the predicate `mg`.

$$\begin{aligned}&\{\ \{\langle 0 = \oplus\texttt{T}, \texttt{One}\rangle, \langle 0 = \oplus\texttt{P} + \ominus\texttt{B}, \texttt{One}\rangle\},\\ &\quad \{\langle 0 < \oplus\texttt{T}, \texttt{One}\rangle, \langle \oplus = \oplus\texttt{T}, \texttt{One}\rangle, \langle 0 \leq \oplus\texttt{P}, \texttt{One}\rangle, \langle 0 = \oplus\texttt{P} + \ominus\texttt{R} + \ominus\texttt{B}, \texttt{One}\rangle\}\ \}\end{aligned}$$

[7] Of course, the analyser takes care of distinguishing "old" and "new" abstract equations, as usual in many abstract interpretation frameworks. See, for instance, [25] for a detailed discussion.

This new output can be used as the output of the recursive call to mg, in order to recompute the output of the second clause of mg. This gives

$$\{\{\langle 0 < \oplus\mathtt{T}, \mathtt{One}\rangle, \langle 0 \le \oplus\mathtt{P}, \mathtt{One}\rangle, \langle 0 = \oplus\mathtt{P} + \ominus\mathtt{R} + \ominus\mathtt{P1}, \mathtt{One}\rangle, \langle \oplus = \oplus\mathtt{T} + \ominus\mathtt{T1}, \mathtt{One}\rangle\}\}$$
$$\uplus$$
$$\begin{array}{l}\{\ \{\langle 0 = \oplus\mathtt{T1}, \mathtt{One}\rangle, \langle 0 = \oplus\mathtt{P1} + \ominus\mathtt{B}, \mathtt{One}\rangle\},\\ \quad \{\langle 0 < \oplus\mathtt{T1}, \mathtt{One}\rangle, \langle \oplus = \oplus\mathtt{T1}, \mathtt{One}\rangle, \langle 0 \le \oplus\mathtt{P1}, \mathtt{One}\rangle, \langle 0 = \oplus\mathtt{P1} + \ominus\mathtt{R} + \ominus\mathtt{B}, \mathtt{One}\rangle\}\ \}\end{array}$$

for the program point after the recursive call. Restricting this to the head variables gives

$$\begin{array}{l}\{\ \{\langle 0 < \oplus\mathtt{T}, \mathtt{One}\rangle, \langle \oplus = \oplus\mathtt{T}, \mathtt{One}\rangle, \langle 0 \le \oplus\mathtt{P}, \mathtt{One}\rangle, \langle 0 = \oplus\mathtt{P} + \ominus\mathtt{R} + \ominus\mathtt{B}, \mathtt{One}\rangle\}\\ \quad \{\langle 0 < \oplus\mathtt{T}, \mathtt{One}\rangle, \langle \oplus = \oplus\mathtt{T}, \mathtt{One}\rangle, \langle 0 \le \oplus\mathtt{P}, \mathtt{One}\rangle,\\ \quad\ \langle 0 = \oplus\mathtt{P} + \ominus\mathtt{R} + \ominus\mathtt{B}, \mathtt{One}\rangle, \langle 0 \le \oplus\mathtt{R} + \oplus\mathtt{B}, \mathtt{One}\rangle\}\ \}\end{array}$$

The union of this with the previously computed output of mg gives the updated output

$$\begin{array}{l}\{\ \{\langle 0 = \oplus\mathtt{T}, \mathtt{One}\rangle, \langle 0 = \oplus\mathtt{P} + \ominus\mathtt{B}, \mathtt{One}\rangle\},\\ \quad \{\langle 0 < \oplus\mathtt{T}, \mathtt{One}\rangle, \langle \oplus = \oplus\mathtt{T}, \mathtt{One}\rangle, \langle 0 \le \oplus\mathtt{P}, \mathtt{One}\rangle, \langle 0 = \oplus\mathtt{P} + \ominus\mathtt{R} + \ominus\mathtt{B}, \mathtt{One}\rangle\}\\ \quad \{\langle 0 < \oplus\mathtt{T}, \mathtt{One}\rangle, \langle \oplus = \oplus\mathtt{T}, \mathtt{One}\rangle, \langle 0 \le \oplus\mathtt{P}, \mathtt{One}\rangle,\\ \quad\ \langle 0 = \oplus\mathtt{P} + \ominus\mathtt{R} + \ominus\mathtt{B}, \mathtt{One}\rangle, \langle 0 \le \oplus\mathtt{R} + \oplus\mathtt{B}, \mathtt{One}\rangle\}\ \}\end{array}$$

The process can be repeated until it gives the following set of abstract store (multi-store) as the fixpoint for the computation of the output of mg.

$$\begin{array}{l}\{\ \{\langle 0 = \oplus\mathtt{T}, \mathtt{One}\rangle, \langle 0 = \oplus\mathtt{P} + \ominus\mathtt{B}, \mathtt{One}\rangle\},\\ \quad \{\langle 0 < \oplus\mathtt{T}, \mathtt{One}\rangle, \langle \oplus = \oplus\mathtt{T}, \mathtt{One}\rangle, \langle 0 \le \oplus\mathtt{P}, \mathtt{One}\rangle, \langle 0 = \oplus\mathtt{P} + \ominus\mathtt{R} + \ominus\mathtt{B}, \mathtt{One}\rangle\}\\ \quad \{\langle 0 < \oplus\mathtt{T}, \mathtt{One}\rangle, \langle \oplus = \oplus\mathtt{T}, \mathtt{One}\rangle, \langle 0 \le \oplus\mathtt{P}, \mathtt{One}\rangle,\\ \quad\ \langle 0 = \oplus\mathtt{P} + \ominus\mathtt{R} + \ominus\mathtt{B}, \mathtt{One}\rangle, \langle 0 \le \oplus\mathtt{R} + \oplus\mathtt{B}, \mathtt{One}\rangle\}\\ \quad \{\langle 0 < \oplus\mathtt{T}, \mathtt{One}\rangle, \langle \oplus = \oplus\mathtt{T}, \mathtt{One}\rangle, \langle 0 \le \oplus\mathtt{P}, \mathtt{One}\rangle,\\ \quad\ \langle 0 = \oplus\mathtt{P} + \ominus\mathtt{R} + \ominus\mathtt{B}, \mathtt{One}\rangle, \langle 0 \le \oplus\mathtt{R} + \oplus\mathtt{B}, \mathtt{One}\rangle, \langle 0 \le \oplus\mathtt{R} + \oplus\mathtt{B}, \mathtt{Any}\rangle\}\ \}\end{array}$$

As the fourth abstract store in the above multi-store subsumes the third store, we can simplify the output description of mg to

$$\begin{array}{l}\{\ \{\langle 0 = \oplus\mathtt{T}, \mathtt{One}\rangle, \langle 0 = \oplus\mathtt{P} + \ominus\mathtt{B}, \mathtt{One}\rangle\},\\ \quad \{\langle 0 < \oplus\mathtt{T}, \mathtt{One}\rangle, \langle \oplus = \oplus\mathtt{T}, \mathtt{One}\rangle, \langle 0 \le \oplus\mathtt{P}, \mathtt{One}\rangle, \langle 0 = \oplus\mathtt{P} + \ominus\mathtt{R} + \ominus\mathtt{B}, \mathtt{One}\rangle\}\\ \quad \{\langle 0 < \oplus\mathtt{T}, \mathtt{One}\rangle, \langle \oplus = \oplus\mathtt{T}, \mathtt{One}\rangle, \langle 0 \le \oplus\mathtt{P}, \mathtt{One}\rangle,\\ \quad\ \langle 0 = \oplus\mathtt{P} + \ominus\mathtt{R} + \ominus\mathtt{B}, \mathtt{One}\rangle, \langle 0 \le \oplus\mathtt{R} + \oplus\mathtt{B}, \mathtt{One}\rangle, \langle 0 \le \oplus\mathtt{R} + \oplus\mathtt{B}, \mathtt{Any}\rangle\}\ \}\end{array}$$

Now, the reader may go back to the motivating discussion of Section 2, and check that the analysis is accurate enough to support every optimization step applied to the program of Example 2.1.
Finally, observe that a systematic normalization (removing redundancies through the ordering relation) may be applied at each step of the analysis to avoid the growth of abstract multi-stores.

4.5. Discussion

To conclude the section, it is instructive to point out a limitation of the domain LSign. On the one hand, consider the constraint store $\theta_1 = \{3 = x, 0 < x\}$ and its LSign abstraction $\beta_1 = \{\langle \oplus = \oplus x, \mathtt{One}\rangle, \langle 0 < \oplus x, \mathtt{One}\rangle\}$. It is easy to see that θ_1 is satisfiable (i.e. $\mathtt{Cis\_sat}(\theta_1)$ is true). Also we have that $\mathtt{Ais\_sat}(\beta_1)$ is true, and so we can make the same conclusion from the abstract domain. On the other hand, consider the constraint store $\theta_2 = \{3 = x, 2 < x\}$ and its LSign abstraction $\beta_2 = \{\langle \oplus = \oplus x, \mathtt{One}\rangle, \langle \oplus < \oplus x, \mathtt{One}\rangle\}$. It is easy to see that $\mathtt{Cis\_sat}(\theta_2)$ is true, however $\mathtt{Ais\_sat}(\beta_2)$ is false. While this does not violate the specification of `Ais_sat`, it means that the `LSign` domain is not able to approximate the store θ_2 sufficiently accurately for applications that need to reason about satisfiability in the abstract domain. Similar examples can be constructed to show loss of accuracy for conditional satisfiability. This suggests the need for a finer analysis than signs and the natural solution is to use intervals instead of signs to approximate numbers. This is explored further in the next section.

5. The Abstract Domain `LInt`

The abstract domain `LInt`, generalizes the domain `LSign` by using intervals instead of signs to abstract coefficients. Technically, the main difficulty arises because `LInt` is an infinite domain unlike `LSign`. This requires the definition of one more abstract operation, viz. the *widening* operation.

The section is organized as follows. Section 5.1 introduces the abstract objects of the domain `LInt`. Section 5.2 briefly presents the operations and applications of the domain. The power domain and the widening are presented in Section 5.3. Section 5.4 concludes by discussing how `LInt` addresses the limitations of `LSign`.

5.1. Abstract Objects and Concretization

We consider the set of real numbers $\Re$ and its extension $\Re^\infty = \Re \cup \{-\infty, \infty\}$. The ordering on $\Re$ is extended to $\Re^\infty$, as usual. We also consider a finite subset $\mathcal{F}$ of $\Re$ containing 0 and the extension of $\mathcal{F}$ viz. $\mathcal{F}^\infty = \mathcal{F} \cup \{-\infty, \infty\}$. In practice, $\mathcal{F}$ is a set of floating point numbers or rational numbers used in the implementation.

The first key idea is the notion of an abstract constraint which abstracts a concrete constraint by replacing each coefficient by an interval over $\mathcal{F}^\infty$.

Definition 5.1. [Intervals] An *interval* is an element of

$$\begin{array}{rll} \mathtt{Intv} = & \{\ [a_l, a_r] & \mid\ a_l, a_r \in \mathcal{F} \ \wedge\ a_l \leq a_r\} \ \cup \\ & \{\ (a_l, a_r] & \mid\ a_l \in \mathcal{F} \cup \{-\infty\} \ \wedge\ a_r \in \mathcal{F} \ \wedge\ a_l < a_r\} \ \cup \\ & \{\ [a_l, a_r) & \mid\ a_l \in \mathcal{F} \ \wedge\ a_r \in \mathcal{F} \cup \{+\infty\} \ \wedge\ a_l < a_r\} \ \cup \\ & \{\ (a_l, a_r) & \mid\ a_l \in \mathcal{F} \cup \{-\infty\} \ \wedge\ a_r \in \mathcal{F} \cup \{+\infty\} \ \wedge\ a_l < a_r\}. \end{array}$$

Intervals are denoted by the letter s, possible subscripted. The monotone concretization function $Cc : \mathtt{Intv} \rightarrow 2^{\Re}$ is as expected. For instance, $Cc([a_l, a_r)) = \{c \mid c \in \Re \ \wedge\ a_l \leq c < a_r\}$.

We assume the existence of functions $\mathtt{left} : \mathtt{Intv} \rightarrow \mathcal{F}^\infty$ and $\mathtt{right} : \mathtt{Intv} \rightarrow \mathcal{F}^\infty$ which return respectively the left and right endpoint of an interval. We also assume the existence of Boolean functions `open_left` and `open_right` which indicate whether an interval is open to the left or to the right. In addition, it is convenient to define two relations $\mathtt{increase} : \mathtt{Intv} \times \mathtt{Intv}$ and $\mathtt{decrease} : \mathtt{Intv} \times \mathtt{Intv}$ as follows:

$$\begin{array}{l} \mathtt{increase}(s_1, s_2) \Leftrightarrow \exists c_1 \in Cc(s_1)\ \forall c_2 \in Cc(s_2) : c_1 > c_2 \\ \mathtt{decrease}(s_1, s_2) \Leftrightarrow \exists c_1 \in Cc(s_1)\ \forall c_2 \in Cc(s_2) : c_1 < c_2 \end{array}$$

The ordering on Intv is defined by $s_1 \sqsubseteq s_2 \Leftrightarrow \neg\texttt{increase}(s_1, s_2) \wedge \neg\texttt{decrease}(s_1, s_2)$, and it satisfies the monotonicity criterion $s_1 \sqsubseteq s_2 \Rightarrow Cc(s_1) \subseteq Cc(s_2)$.

We also assume operations like $+, -, \times, \sqcup, \sqcap$ on intervals which are consistent approximations of the corresponding operations on $\Re^\infty$. For instance, $c_1 \in Cc(s_1) \wedge c_2 \in Cc(s_2) \Rightarrow c_1 + c_2 \in Cc(s_1 + s_2)$.

We say that an interval is *positive* (*negative*, resp.) if it contains only positive (negative, resp.) real numbers. Moreover, we say that an interval s is *zero* if $s = [0, 0]$. These properties are represented by the boolean functions `pos, neg, zer`. We also define the boolean function `contains_pos` (`contains_neg`, resp.) to indicate whether an interval contains at least one positive (negative resp.) number. Moreover, we define the boolean function `contains_zer` to indicate whether an interval contains the number zero. The definitions of operators, abstract constraints, multiplicities, abstract constraints with multiplicities, and abstract stores of the domain are completely parallel to those in the domain `LSign`. The only difference is that signs are replaced by intervals in the abstract constraints. We limit ourselves to some examples to explain these concepts in the domain `LInt`.

Example 5.1. The abstract constraint $(-\infty, +\infty) = [1,1]\texttt{P} + [1,1]\texttt{R}$ represents both the constraint $3 = \texttt{P} + \texttt{R}$ and $-3 = \texttt{P} + \texttt{R}$ but not the constraint $3 = 2\texttt{P} + 3\texttt{R}$.
The abstract constraint with multiplicity $\langle\, (-\infty, +\infty) = [1,1]\texttt{P}+[1,1]\texttt{R},\ \texttt{One}\,\rangle$ represents only multisets of size 1, e.g., $\{3 = \texttt{P}+\texttt{R}\}$. $\langle\, 0 \leq [1.01, +\infty)\texttt{P}+(-\infty, -1]\texttt{R},\ \texttt{Any}\,\rangle$ represents multisets of any size, e.g., $\emptyset$, $\{0 \leq 3\texttt{P} - \texttt{R}\}$ and $\{0 \leq 3\texttt{P} - \texttt{R}, 0 \leq 2\texttt{P} - 3\texttt{R}\}$. The abstract store $\beta = \{\langle\, (-\infty, +\infty) = [1,1]\texttt{P} + [1,1]\texttt{R},\ \texttt{One}\,\rangle\ ,\ \langle\, 0 \leq [1.01, +\infty)\texttt{P} + (-\infty, -1]\texttt{R},\ \texttt{Any}\,\rangle\}$ represents constraint stores with at least one constraint, and their equivalence classes. For example $\{3 = \texttt{P} + \texttt{R}\} \in Cc_i(\beta)$ and $\{3 = \texttt{P} + \texttt{R}\ , 0 \leq 2\texttt{P} - 3\texttt{R}\} \in Cc_i(\beta)$. Further, $\{3 = \texttt{P} + \texttt{R}\ ,\ 9 \leq 5\texttt{P}\} \in Cc(\beta)$ because $\{3 = \texttt{P} + \texttt{R}\ ,\ 9 \leq 5\texttt{P}\} \leftrightarrow \{3 = \texttt{P} + \texttt{R}\ , 0 \leq 2\texttt{P} - 3\texttt{R}\}$.

5.2. Operations and Applications

The algorithms for various operations (ordering, addition of a constraint, upper bound and projection) as well as the applications (satisfiability and conditional satisfiability) of `LInt` are for the most part identical to the corresponding algorithms in `LSign`. The only difference is that the various operations on signs are replaced by the corresponding operations on intervals. For example, a test $\sigma[v] = \oplus$ would be replaced by the test $\texttt{pos}(\sigma[v])$, while an assignment $\sigma[v] := 0$ would be replaced by the assignment $\sigma[v] := [0, 0]$. The only algorithm that undergoes a non-trivial change is the algorithm for operation `Asplit_top`. It can be more precise due to the more precise information available about the coefficient of the variable being eliminated. The algorithm for `Asplit_top` in `LInt` is given in Figure 5.1.

5.3. The Power Domain $2^{\texttt{LInt}}$ and Widening

Just like `LSign`, the upper bound operation in `LInt` may not be sufficiently accurate to perform a practical analysis of CLP programs. It may become necessary to move to the power domain $2^{\texttt{LInt}}$ in order to get the required accuracy. The definition of the various concepts in the domain $2^{\texttt{LInt}}$ is completely parallel the corresponding

```
Asplit_top(β, v) {
if β = ∅ return ∅;
else let   β = {γ} ∪ β′ ∧ γ = ⟨σ, μ⟩ ∉ β′
    γ⁰ = ⟨σ⁰, μ ⊔ ZeroOrOne⟩
    γ⁺ = ⟨σ⁺, μ ⊔ ZeroOrOne⟩
    γ⁻ = ⟨σ⁻, μ ⊔ ZeroOrOne⟩
    σ⁰ = σ except that σ⁰[v] = [0, 0]
    σ⁺ = σ except that σ⁺[v] = σ[v] ⊓ (0, +∞)
    σ⁻ = σ except that σ⁻[v] = σ[v] ⊓ (−∞, 0) in
    case not (pos(σ[v]) ∨ neg(σ[v]) ∨ zer(σ[v])):
        β″ := ∅
        if contains_zer(σ[v])
            β″ := β″ ⊎ γ⁰;
        if contains_pos(σ[v])
            β″ := β″ ⊎ γ⁺;
        if contains_neg(σ[v])
            β″ := β″ ⊎ γ⁻;
        return Asplit_top(β′,v) ⨄ β″;
    otherwise:
        return Asplit_top(β′,v) ⊎ γ;
}
```

FIGURE 5.1. Modified `Asplit_top` Algorithm for `LInt`

definitions in the domain $2^{\texttt{LSign}}$. However, as $2^{\texttt{LInt}}$ is an infinite domain, it becomes necessary to introduce a *widening* operator[8]. The use of widening operators was proposed in [6], further discussed in [7], and has been used in domains such as type graphs for Prolog [40].

Naive widenings for $2^{\texttt{LInt}}$ induce a substantial loss of precision, even on small examples. The main contribution of this section is to show how `LSign` can be used to guide the widening of `LInt`. The basic intuition behind our widening is to try to guess where the constraint stores are growing. A fundamental observation here is the fact that, in general, the growth preserves the shape of the `LInt` constraint store when it is viewed as a `LSign` constraint.

The definition of the widening operator proceeds systematically at each level of abstract objects. For each abstract object, it is convenient to define the *shape* of that object, which is obtained by replacing intervals in that object by the corresponding signs.

Definition 5.2. [Shape and Widening of Intervals] The shape of an interval s is defined as

$$\texttt{shape}(s) = \begin{cases} \oplus \text{ if } \texttt{pos}(s) \\ \ominus \text{ if } \texttt{neg}(s) \\ 0 \;\text{ if } \texttt{zer}(s) \\ \top \text{ otherwise} \end{cases}$$

[8]In fact, it is necessary to define a widening for the `LInt` domain as well, but we shall focus on the $2^{\texttt{LInt}}$ domain, as it is used in our compiler.

Let s_{old} and s_{new} be intervals s.t. $\mathtt{shape}(s_{new}) = \mathtt{shape}(s_{old})$. Then

$$
\begin{array}{l}
s_{new} \nabla s_{old} = s' \text{ such that} \\
\quad \mathtt{left}(s') = \begin{cases} 0 & \text{if } \mathtt{decrease}(s_{new}, s_{old}) \wedge \mathtt{pos}(s_{old}) \\ -\infty & \text{if } \mathtt{decrease}(s_{new}, s_{old}) \wedge \neg\mathtt{pos}(s_{old}) \\ \mathtt{left}(s_{old}) & \text{otherwise} \end{cases} \\
\quad \mathtt{open\_left}(s') = \mathtt{decrease}(s_{new}, s_{old}) \vee \mathtt{open\_left}(s_{old}) \\
\quad \mathtt{right}(s') = \begin{cases} 0 & \text{if } \mathtt{increase}(s_{new}, s_{old}) \wedge \mathtt{neg}(s_{old}) \\ \infty & \text{if } \mathtt{increase}(s_{new}, s_{old}) \wedge \neg\mathtt{neg}(s_{old}) \\ \mathtt{right}(s_{old}) & \text{otherwise} \end{cases} \\
\quad \mathtt{open\_right}(s') = \mathtt{increase}(s_{new}, s_{old}) \vee \mathtt{open\_right}(s_{old})
\end{array}
$$

The widening is defined only for intervals that have the same shape, an can be seen as an extension of the widening on interval domains as defined in [7]. We discuss how the left endpoint of the intervals is widened. The right endpoint is similar. If the new interval s_{new} is growing at the left endpoint relative to the old interval s_{old}, the widened left endpoint is set to the minimum possible value that does not alter the shape of the interval. This means that if a positive interval is growing at the left endpoint, its widened left endpoint is set to 0 and made open. If a non-positive interval is growing at the left endpoint, its widened left endpoint is set to $-\infty$. Otherwise the left endpoint is not changed. The reason for distinguishing between positive and non-positive intervals is to make sure that the widened interval has the same shape as the original intervals. Also the intervals are not made smaller by the widening and so the following lemma applies.

Lemma 5.1. Let s_{old} and s_{new} be intervals s.t. $\mathtt{shape}(s_{new}) = \mathtt{shape}(s_{old})$. *Then*
(i) $\mathtt{shape}(s_{old}) = \mathtt{shape}(s_{new} \nabla s_{old})$.
(ii) $c \in Cc(s_{old}) \vee c \in Cc(s_{new}) \Rightarrow c \in Cc(s_{new} \nabla s_{old})$.

The notion of shape and widening can be easily lifted to abstract constraint with multiplicities, abstract stores, and abstract multistores (see [34] for complete details). For each such abstract object, normalization operations are introduced that avoid redundancies (with respect to the shape), leading to the following definitions:

Definition 5.3. [Normalized Abstract Multistore] An abstract multistore α is said to be normalized if

- $\forall \beta \in \alpha : \forall \gamma_1, \gamma_2 \in \beta : \mathtt{shape}(\gamma_1) \neq \mathtt{shape}(\gamma_2)$;
- $\forall \beta_1, \beta_2 \in \alpha : \mathtt{shape}(\beta_1) \neq \mathtt{shape}(\beta_2)$.

Given an abstract multistore α, it is possible to transform it into a corresponding normalized multistore $\mathtt{normal}(\alpha)$ such that $\theta \in Cc(\alpha) \Rightarrow \theta \in Cc(\mathtt{normal}(\alpha))$.

Definition 5.4. [Shape and Widening of Abstract Multistores] The shape of an normalized abstract multistore α is defined as $\mathtt{shape}(\alpha) = \{\mathtt{shape}(\beta) \mid \beta \in \alpha\}$.
Let α_{old} and α_{new} be normalized abstract multistores. Then

$$
\begin{array}{l}
\alpha_{new} \nabla \alpha_{old} \\
= \begin{cases} \alpha_{old} & \text{if } \alpha_{new} \sqsubseteq \alpha_{old} \\ \{\beta_{old} \mid \beta_{old} \in \alpha_{old},\ \mathtt{shape}(\beta_{old}) \notin \mathtt{shape}(\alpha_{new})\} \cup & \text{otherwise} \\ \{\beta_{new} \mid \beta_{new} \in \alpha_{new},\ \mathtt{shape}(\beta_{new}) \notin \mathtt{shape}(\alpha_{old}\} \cup & \\ \{\beta_{new} \nabla \beta_{old} \mid \beta_{new} \in \alpha_{new},\ \beta_{old} \in \alpha_{old},\ \mathtt{shape}(\beta_{new}) = \mathtt{shape}(\beta_{old})\} & \end{cases}
\end{array}
$$

Intuitively, the widening for abstract multistores is a generalization of the upper bound operation. The abstract stores belonging to α_{old} and α_{new} need to be added to $\alpha_{new}\nabla\alpha_{old}$, however if there are two abstract stores with the same shape, their widening needs to be computed first. The following lemma states that the widening for abstract multistores preserves the normalized form and that the widened multistore's concretization includes the concretizations of α_{old} and α_{new}.

Lemma 5.2. Let α_{old} and α_{new} be normalized abstract multistores. Then
(i) $\alpha_n\nabla\alpha_{old}$ is normalized.
(ii) $\theta \in Cc(\alpha_{old}) \vee \theta \in Cc(\alpha_{new}) \Rightarrow \theta \in Cc(\alpha_{new}\nabla\alpha_{old})$.

The following theorem states the correctness of the operator ∇.

Theorem 5.1 Widening. Operation ∇ is a widening operator.

To conclude the section, we show the resulting output description for the mortgage program `mg/4` of Example 2.1 on this domain.

Example 5.2. [Computation of $2^{\texttt{LInt}}$ Output Description for `mg/4`] The abstract substitution describing the output of the first clause is

$$\{\{\langle[0,0] = [1,1]\texttt{T}, \texttt{One}\rangle, \langle[0,0] = [1,1]\texttt{P} + [-1,-1]\texttt{B}, \texttt{One}\rangle\}\}$$

At the end of the analysis of both clauses, requiring the application of the widening operation, the computation leads to the following multistore, which is the fixpoint of the computation and represents the output of the predicate `mg`:

$$\begin{array}{l}
\{\ \{\langle[0,0] = [1,1]\texttt{T}, \texttt{One}\rangle, \langle[0,0] = [1,1]\texttt{P} + [-1,-1]\texttt{B}, \texttt{One}\rangle\},\\
\quad\{\langle[0,0] < [1,1]\texttt{T}, \texttt{One}\rangle, \langle[1,1] = [1,1]\texttt{T}, \texttt{One}\rangle,\\
\quad\ \ \langle[0,0] \leq [1,1]\texttt{P}, \texttt{One}\rangle, \langle[0,0] = [1.01, 1.01]\texttt{P} + [-1,-1]\texttt{R} + [-1,-1]\texttt{B}, \texttt{One}\rangle\}\\
\quad\{\langle[0,0] < [1,1]\texttt{T}, \texttt{One}\rangle, \langle[0,0] \leq [1,1]\texttt{P}, \texttt{One}\rangle, \langle[2,3] = [1,1]\texttt{T}, \texttt{One}\rangle,\\
\quad\ \ \langle[0,0] = [1.0201, 103.0301]\texttt{P} + [-303.01, -2.01]\texttt{R} + [-100,-1]\texttt{B}, \texttt{One}\rangle,\\
\quad\ \ \langle[0,0] \leq [1,201]\texttt{R} + [1,100]\texttt{B}, \texttt{One}\rangle, \langle\ [0,0] \leq [1,201]\texttt{R} + [1,100]\texttt{B}, \texttt{Any}\rangle\},\\
\quad\{\langle[0,0] < [1,1]\texttt{T}, \texttt{One}\rangle, \langle[0,0] \leq [1,1]\texttt{P}, \texttt{One}\rangle, \langle[3,+\infty) = [1,1]\texttt{T}, \texttt{One}\rangle,\\
\quad\ \ \langle[0,0] = [1.030301, +\infty)\texttt{P} + (-\infty, -3.0301]\texttt{R} + (-\infty,-1]\texttt{B}, \texttt{One}\rangle,\\
\quad\ \ \langle[0,0] \leq [1,201]\texttt{R} + [1,100]\texttt{B}, \texttt{One}\rangle, \langle\ [0,0] \leq [1,201]\texttt{R} + [1,100]\texttt{B}, \texttt{Any}\rangle,\\
\quad\ \ \langle[0,0] \leq [1.01, +\infty)\texttt{P} + (-\infty,-1]\texttt{R}, \texttt{One}\rangle,\\
\quad\ \ \langle[0,0] \leq [1.01, +\infty)\texttt{P} + (-\infty,-1]\texttt{R}, \texttt{Any}\rangle\}\ \}
\end{array}$$

Comparing this multistore with the one shown in Example 4.17, we may observe how more accurate the analysis using $2^{\texttt{LInt}}$ is with respect to the analysis on $2^{\texttt{LSign}}$. For instance, in the first two stores (that correspond strictly, in the two examples), there is no loss of information when using $2^{\texttt{LInt}}$ with respect to the concrete computation, whereas $2^{\texttt{LSign}}$ analysis immediately looses track of any numeric value.

5.4. Discussion

The domain `LInt` enables us to overcome the limitation of the domain `LSign` that was pointed out in the previous section. Consider the constraint store $\theta = \{3 = x, 2 < x\}$. Its `LSign` abstraction is $\beta = \{\langle\oplus = \oplus x, \texttt{One}\rangle, \langle\oplus < \oplus x, \texttt{One}\rangle\}$. While θ is satisfiable, $\texttt{Ais\_sat}(\beta)$ is false because the abstract store obtained by projecting all the variables is $\{\langle\top < 0, \texttt{One}\rangle\}$. Moving to the domain `LInt`, the abstraction of θ is

Program	Mode	Description
Integer	f	Is 25000 an integer?
	u	Generate $0 \ldots 250$
Exp	fu	Compute 2^{25}
	uf	Compute $\lg 2^{25}$
	uu	Generate $(0,1) \ldots (25, 2^{25})$
Sum	fu	Compute $0 + 1 + \ldots + 500$
	uf	Find N s.t. $0 + 1 + \ldots + N = 125250$
	uu	Generate $(0,0) \ldots (500, 125250)$
Fibonacci	fu	Compute 15^{th} Fibonacci number
	uf	Find N s.t. 987 is N^{th} Fibonacci number
	uu	Generate $(0,1) \ldots (15, 987)$
Mortgage (Linear)	fffu (1)	Principal = 100, Time = 50; find Balance
	fffu (2)	Principal = 200, Time = 100; find Balance
	fufu (1)	Principal = 100, Time = 0 ... 50; find Balance
	fufu (2)	Principal = 200, Time = 0 ... 100; find Balance
Mortgage (Nonlinear)	ffffu (1)	Principal = 100, Time = 50 ; find Balance
	ffffu (2)	Principal = 200, Time = 100; find Balance
	fuffu (1)	Principal = 100, Time = 0 ... 50; find Balance
	fuffu (2)	Principal = 200, Time = 0 ... 100; find Balance
Ode-Euler	ffffu	Compute final y value
	fufff	Compute initial y value
	fuffu	Relate initial and final y values
Triangular, 2000		Solve N equations, N = 2000
4000		Solve N equations, N = 4000
8000		Solve N equations, N = 8000

TABLE 6.1. Test Programs: Description of Usage in Various Modes.

$\beta' = \{\langle [3,3] = [1,1]x, \mathtt{One}\rangle, \langle [2,2] < [1,1]x, \mathtt{One}\rangle\}$. Projecting all the variables gives the abstract store $\{\langle [-1,-1] < 0, \mathtt{One}\rangle\}$ and so $\mathtt{Ais\_sat}(\beta')$ is true, which represents the concrete operation more accurately. Similar examples can be constructed for the other abstract applications such as conditional satisfiability.

6. Preliminary Experimental Results

The purpose of this section is to give very preliminary evidence that the domain $2^{\mathtt{LSign}}$ is sufficiently precise to perform the intended optimizations. It shows that an optimizing compiler can indeed use the domain and perform the intended optimizations on small programs and it gives the magnitude of the speed-ups. The section also gives some preliminary information on the price to pay when using $2^{\mathtt{LInt}}$ instead of $2^{\mathtt{LSign}}$. The compilation times given in this section should be interpreted with care. The compiler is not incremental and restarts the analysis as soon as a reordering takes place. Obviously, a practical compiler should be incremental and the techniques presented in [17] should be helpful here to obtain a fast implementation. Our benchmarks are relatively small, since most large $\mathtt{CLP}(\Re_{Lin})$ programs use first-order terms in addition to linear constraints and our analyzer is not able to handle them at this stage. Most of the programs are also multi-directional and they are run with various modes: `u` stands for an unconstrained variable while `f` stands for a fixed variable. Program `Integer(N)` is used to verify if 25000 is an integer and to

Program	Mode	Unopt. (1)	Opt. (2)	Speedup (1)/(2)	Annotation	Bypass?
Integer	f	1380	1280	1.08	REF	×
	u	4810	280	17.18	REO,REF,REM	√
Exp	fu	10	10	1.00	REO,REF	√
	uf	100	10	10.00	REO,REF,REM	√
	uu	120	10	12.00	REO,REF,REM	√
Sum	fu	200	40	5.00	REO,REF	√
	uf	19220	17270	1.11	REO, REM	×
	uu	31490	2180	14.44	REO,REF,REM	√
Fibonacci	fu	720	240	3.00	REO,REF	√
	uf	8570	1950	4.39	REO, REF	×
	uu	7880	810	9.73	REO, REF	×
Mortgage (Linear)	fffu (1)	410	390	1.05	REF	×
	fffu (2)	30	20	1.50	REF	×
	fufu (1)	880	420	2.10	REO,REF,REM	√
	fufu (2)	870	70	12.43	REO,REF,REM	√
Mortgage (Nonlinear)	ffffu (1)	800	690	1.16	REF	×
	ffffu (2)	50	40	1.25	REF	×
	fuffu (1)	1750	1100	1.59	REO,REF,REM	√
	fuffu (2)	1590	140	11.36	REO,REF,REM	√
Ode-Euler	ffffu	1260	1120	1.13	REF	×
	fufff	1540	1440	1.07	REO,REF	×
	fuffu	1330	1210	1.10	REO,REF	×
Triangular, 2000		520	130	4.00	REF,REO	√
4000		1660	210	7.90	REF,REO	√
8000		4490	290	15.48	REF,REO	√
Ar. Mean				5.68		

TABLE 6.2. Comparison of Running Times in msec.: Optimized vs. Unoptimized using $2^{\mathtt{LSign}}$-analysis.

generate the first 250 integers. The next three programs, `Exp(N,E)`, `Sum(N,S)` and `Fibonacci(N,F)` are programs that compute $2^{\mathtt{N}}$, the sum of the first `N` integers, and the $\mathtt{N}^{th}$ Fibonacci number, respectively. `Mortgage` relates the various parameters of a mortgage computation, and we have two versions of it. The first is the running example used in the paper. The second is a syntactically nonlinear version which has the interest rate as an argument. We have used an interest rate of 1% per month and a monthly repayment of 2 units; the final balance is unconstrained. The value of the principal and time period vary to illustrate various tradeoffs in the optimizations. `Ode-Euler` [29] is a program solving the ordinary differential equation $y' = t$. `Triangular` is a benchmark that involves simultaneously solving a sparse system of `N` equations, subsystems of which are in upper triangular form. Table 6.1 summarizes the usage modes of the various test programs.

The benchmarks are used to compare the optimizing compiler (Opt.) with the standard compiler (Unopt.). Both compilers generate code for the same runtime system which is a WAM-based system with special instructions for tests and assignments. The architecture of the system was described in [39]. The runtime system uses infinite precision integers to guarantee numerical stability. Table 6.2 reports the computation times (in milliseconds on a SUN-SPARC-10) for the optimized and unoptimized versions of the benchmarks for various modes on $2^{\mathtt{LSign}}$. It also

Program	Mode	Reord. (1)	Removal (2)	Refin. (3)	Total $2^{\texttt{LSign}}$ (4) $= \Sigma_{i=1..3}(i)$	$2^{\texttt{LInt}}$ (5)	Ratio (5)/(4)
Integer	f	120	20	10	150	800	5.33
	u	400	10	10	420	2150	5.12
Exp	fu	510	10	10	530	4110	7.75
	uf	690	10	10	710	5140	7.24
	uu	980	10	20	1010	7020	6.95
Sum	fu	1200	10	10	1220	8060	6.61
	uf	1470	20	20	1510	10340	6.88
	uu	1530	10	10	1550	11180	7.21
Fibonacci	fu	4280	30	20	4330	43910	10.14
	uf	7900	90	80	8070	120750	14.96
	uu	6150	70	50	6270	76610	12.22
Mortgage (Linear)	fffu	1420	20	20	1460	14410	9.87
	fufu	3060	30	10	3100	42420	13.68
Mortgage (Nonlinear)	ffffu	3350	50	40	3440	25070	7.29
	fuffu	9480	50	30	9560	62180	6.50
Ode-Euler	ffffu	1680	40	10	1730	11110	6.42
	fufff	1650	30	20	1700	11070	6.51
	fuffu	1140	20	20	1180	7200	6.10
Triang., 2000		360	20	10	390	1350	3.46
4000		480	30	10	520	1510	2.90
8000		2320	80	50	2450	3920	1.60
Ar. Mean							7.39

TABLE 6.3. Optimization Times in Milliseconds: $2^{\texttt{LSign}}$ and $2^{\texttt{LInt}}$.

gives the ratio of the unoptimized execution time to the optimized execution time (*speedup*). The table also specifies which optimizations have been performed. `REO` indicates that constraints were reordered, `REF` indicates that constraints were refined to tests or assignments and `REM` indicates that (redundant) constraints were removed from the program. The final column in the table indicates whether the optimizations enabled the execution to bypass the constraint solver entirely (i.e perform only tests and assignments in the engine and interface). A $\surd$ indicates that the unoptimized program utilized the constraint solver to perform constraint solving, while the optimized program bypassed the constraint solver entirely. For all these programs, $2^{\texttt{LSign}}$-analysis is sufficient to get maximal precision for reordering, and passing to $2^{\texttt{LInt}}$ does not bring any improvement with that respect. However, it is sufficient to slightly modify their code (e.g. moving the basic case from 0 to 1) to see that $2^{\texttt{LInt}}$ still supports the optimization while `LSign` looses most of its accuracy.

The speedups vary from 1.00 on one of the `Exp` queries to 15.48 on the `Triangular` program, when 8000 equations are solved. Seven programs exhibit speedups of at least 10, while eleven programs exhibit speedups lower than 2. The average speedup observed was 5.68.

Finally, we investigated the price the pay to move from $2^{\texttt{LSign}}$ to $2^{\texttt{LInt}}$. Table 6.3 gives the optimization time (in milliseconds) for $2^{\texttt{LSign}}$, separately for each of the three phases, as well as for the total of the three phases; then, it compares the optimization times for the benchmarks when $2^{\texttt{LInt}}$ is used in the analyzer instead of $2^{\texttt{LSign}}$. For our benchmarks, the average penalty paid by the $2^{\texttt{LInt}}$ analysis over the $2^{\texttt{LSign}}$ analysis is a factor of 7.39.

7. Conclusions

This paper has studied two abstract domains, `LSign` and `LInt`, which can be used at compile time to determine when it is safe to reorder CLP programs. As far as `LSign` is concerned, the paper redefined the domain of [28] to simplify the correctness proofs. It proposed a new ordering capturing the intended meaning of [28] computable in polynomial time. It proposed a more precise algorithm for projection. Finally, it discussed how `LSign` can be used to detect the conditional satisfiability of constraint stores, an operation which is fundamental to reorder $\texttt{CLP}(\Re_{Lin})$ programs. This operation, which was never discussed previously, was shown to raise some subtle practical issues which are studied at length. As far as the new domain `LInt` is concerned, the paper proposed in particular a precise widening operator. The paper also described some very preliminary evidence showing that the domains are precise enough to perform the intended optimizations on small programs.

There are of course many avenues for future research. Of immediate concern is the development of a fast and incremental optimizing compiler using the technology described in [17]. Equally important are the enhancement of the domains to support first-order terms, and their integration in the generic `Pattern` domain [5] to optimize the interaction with unification constraints. These developments will make it possible to apply the domains on large programs and to give a definitive assessment on their practical value.

Acknowledgments

We are grateful to the anonymous referees for insightful suggestions. Agostino Cortesi is partially supported by the Italian MURST project N. 9701248444-044.

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, G. Janssens, A. Callebaut and B. Demoen. Abstract Interpretation: Towards the Global Optimization of Prolog Programs. In *Proc. 1987 Symp. on Logic Programming*, 192–204, San Francisco, CA, August 1987.
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. *Commun. ACM*, 28(4):412–418, 1990.
5. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of Abstract Domains for Logic Programming. In *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*, 227–239, Portland, OR, January 1994.
6. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conf. Record of Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, January 1977.

7. P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *Proc. of the International Symposium PLILP'92*. LNCS 631, pages 269–295. Springer-Verlag, Berlin, 1992.
8. 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)*,pages 84–96, 1978.
9. S. Debray. Unfold/Fold Transformations and Loop Optimizations of Logic Programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI-88)*, pages 297–307, Atlanta, 1988.
10. V. Dumortier, G. Janssens, M. Bruynooghe and M. Codish. Freeness Analysis in the Presence of Numerical Constraints. In *Tenth International Conference on Logic Programming (ICLP-93)*, 100–115, MIT Press, June 1993.
11. 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 Margherita Ligure, Italy, June 1994.
12. G. Filé and F. Ranzato. Improving abstract interpretations by systematic lifting to the powerset. In *Eleventh International Conference on Logic Programming (ICLP-94)*, Santa Margherita Ligure, Italy, June 1994. The MIT Press.
13. M.L. Fredman and R.E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *JACM*, 34:596–615, July 1987.
14. 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.
15. M.J. Garcia de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564-614, 1996.
16. M. García de la Banda, K. Marriott, P. Stuckey, H. Soondergaard Differential Methods in Logic Program Analysis. *Journal of Logic Programming* Vol 35(1):1-37, 1998.
17. M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey Incremental Analysis of Logic Programs. In *1995 International Conference on Logic Programming*, Japan, June, 1995.
18. J.-L. Imbert, J. Cohen, and M.-D. Weeger. An Algorithm for Linear Constraint Solving: Its Incorporation in a Prolog Meta-Interpreter for CLP. *Journal of Logic Programming* 16(3-4):235–254, 1993.
19. J. Jaffar and J-L. Lassez. Constraint logic programming. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages POPL87*, 111–119, Munich, Germany, January 1987.
20. J. Jaffar, S. Michaylov, P.J. Stuckey, and R. Yap. The CLP($\Re$) language and system. *ACM Trans. on Programming Languages and Systems*, 14(3):339–395, 1992.
21. J. Jaffar, S. Michaylov, P.J. Stuckey, and R. Yap. An Abstract Machine for CLP($\Re$). In *Proc. ACM-SIGPLAN Conf. on Programming Language Design and Implementation (PLDI-92)*, 128–139, 1992, ACM Press.
22. N. Jorgensen, K. Marriott, and S. Michaylov. Some Global Compile Time Optimizations for CLP($\Re$). In *Logic Programming: Proc. 1991 Intl. Symp.*, 420–434, MIT Press, 1991.
23. A.D. Kelly, A.D. Macdonald, K. Marriott, H. Soondergaard, P.J. Stuckey, and R.H C. Yap. An Optimizing Compiler for CLP(R). In *Proc. First Int. Conference on Principles and Practice of Constraint Programming - CP'95*, LNCS, Vol. 976, 222–239, Springer 1995
24. A.D. Kelly, A.D. Macdonald, K. Marriott, P.J. Stuckey, and R.H C. Yap. Ef-

fectiveness of Optimizing Compilation for CLP(R) In *Proc. Joint International Conference and Syposium on Logic Programming*, 37–51, MIT Press, 1996.
25. 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.
26. A. MacDonald, P. Stuckey, and R. Yap. Redundancy of Variables in CLP($\Re$). In *Logic Programming: Proc. 1993 Intl. Symp.*, 75–93, 1993, MIT Press.
27. 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.
28. 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.
29. S. Michaylov and B. Pippin. Optimizing Compilation of Linear Arithmetic in a Class of Constraint Logic Programs. In *Proceedings of the International Symposium on Logic Programming (ILPS-94)*, pages 586–600, Ithaca, NY, November 1994.
30. W. Older and A. Vellino. Extending Prolog with Constraint Arithmetics on Real Intervals. In *Canadian Conf. on Computer & Electrical Engineering*, Ottawa, 1990.
31. G. Puebla and M. Hermenegildo. Implementation of Multiple Specialization in Logic Programs. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 77–87. ACM Press, June 1995.
32. G. Puebla and M. Hermenegildo. Abstract Specialization and its Application to Program Parallelization In *VI International Workshop on Logic Program Synthesis and Transformation*, LNCS-1207, Springer Verlag, 1997.
33. V. Ramachandran and P. Van Hentenryck. LSign Reordered. In Alan Mycroft (Ed.) *Proc. Static Analysis Symposium (SAS-95).* LNCS, Vol. 983, 330–347, Springer Verlag, 1995.
34. V. Ramachandran, P. Van Hentenryck, and A. Cortesi. Abstract Domains for Reordering of `CLP(`$\Re_{Lin}$`)`. Technical Report, CS-97-10, Ca' Foscari University, June 1997. `http://www.dsi.unive.it/~cortesi/paperi/TR_CS_97_10`.
35. P. Refalo, and P.Van Hentenryck. CLP(Rlin) Revised *Proc. Joint International Conference and Symposium on Logic Programming*, 22–36, MIT Press, 1996.
36. V. Ramachandran. An Optimizing Compiler for `CLP(`$\Re_{Lin}$`)`, PhD thesis, Brown University, forthcoming.
37. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series, The MIT Press, Cambridge, Mass., 1989.
38. 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.
39. P. Van Hentenryck and V. Ramachandran. Backtracking without Trailing in CLP($\Re_{Lin}$). *ACM Trans. on Programming Languages and Systems*, 17(4), pp. 635-671, July 1995.
40. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type Analysis of Prolog Using Type Graphs. *Journal of Logic Programming* 22(3): 179-209, 1995.
41. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of `Prop`. *Journal of Logic Programming*, 23(3): 237–278, 1995.