Abstract Domains for Reordering CLP($\Re_{Lin}$) Programs*

V. Ramachandran
Brown University
Providence RI, USA
pvh@cs.brown.edu

P. Van Hentenryck
Brown University
Providence RI, USA
vr@cs.brown.edu

A. Cortesi
Università Ca' Foscari
Venice, Italy
cortesi@unive.it

Abstract

Constraint Logic Programming (CLP), a generalization of logic programming where constraint solving replaces unification as the basic operation of the language, offers many opportunities for optimization. In particular, to address the multi-directional nature of CLP programs, recent optimizing compilers may 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. However, it is important to make sure that reordering does not alter the search space explored by the program to preserve its efficiency and termination properties.

This paper focuses on CLP($\Re_{Lin}$), i.e., constraint logic programming 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 in [38] and further studied in [46]. 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. The paper contains a comprehensive description of the two domains together with their correctness proofs. It also contains some very preliminary evidence showing that the domains are precise enough to perform the intended optimizations on small programs.

General Terms: LANGUAGES, THEORY, VERIFICATION

Categories and Subject Descriptors: D.1.1 : Software, PROGRAMMING TECHNIQUES, Applicative (Functional) Programming. D.1.6 : Software, PROGRAMMING TECHNIQUES, Logic Programming. D.3.4 : Software, PROGRAMMING LANGUAGES, Processors, Compilers. D.3.4 : Software, PROGRAMMING LANGUAGES, Processors, Optimization. F.3.1 : Theory of Computation, LOGICS AND MEANINGS OF PROGRAMS, Specifying and Verifying and Reasoning about Programs.

*A preliminary version of this paper appeared in the 1995 Static Analysis Symposium.

1 Introduction

Constraint logic programming (CLP) [24] 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 linear real constraints (e.g., [25, 51, 8]), integers (e.g., [50]), Booleans (e.g., [6, 8]), and nonlinear real constraints (e.g., [43]). CLP languages preserve some of the traditional advantages of logic programming such as declarative semantics, nondeterminism and multi-directionality, while adding the expressive power of constraint solving over a suitable domain. These features make CLP languages especially suitable for expressing combinatorial search problems. CLP languages have been used to solve problems in diverse areas such as hardware design (eg. circuit verification), finance (eg. options trading), biology (eg. DNA sequencing) and operations research (eg. cutting stock problems).

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 [25, 53, 47], i.e. on constraint logic programs over linear real constraints, and the approach to CLP optimization proposed in [37]. 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., [58, 19, 44, 45]) 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 [38] and further studied in [46]. 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., [14, 15, 18, 20]) and these domains are potentially useful for reordering as well. However, as acknowledged in [20], 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 [46], 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 [48] and deserves an independent treatment.

As far as LSign is concerned, the paper redefines the domain of [38] to simplify the correctness proofs. It proposes a new ordering capturing the intended meaning of [38] and shows that two LSign descriptions can be ordered in polynomial time. It also proposes a more precise algorithm for projection whose essential structure is modeled after [38]. 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 contained in the paper can be found in the Appendix. Only sketches of the proofs are given in the main text.

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 [25] which relates various parameters in a mortgage computation. This program will be used as running example in the paper.

```
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 monthly interest rate of

1%. The most interesting feature of this progam 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 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 not even resort to contraint solving. In the rest of this section, we present how this is achieved by the first pass of the optimizing compiler [48] (see [29, 30] for other optimizing compilers). 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. An assignment is of the form `Var := Exp` which, when executed, assumes that `Var` is an unconstrained variable while `Exp` is a fixed expression (i.e., an expression whose variables are fixed to a value). A test is of the form `Exp1 ?`δ `Exp2`, where $\delta \in \{=, <, \leq, >, \geq, \neq\}$ and where `Exp1` and `Exp2` are fixed expressions. 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 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 as the convex hull domain of [12]. 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, 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 at this point 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, in our running example, the techniques of [13] can be used to transform our final program into a tail-recursive program. Similarly, efficient instructions can be generated for the tests and assignments [26].

3 The Concrete Domain

Abstract interpretation [11] is a general methodology to design static analyses of programs in a systematic way. The basic intuition behind abstract interpretation is to infer some properties of a program, not by executing it on its traditional computation domain (say integers), but rather on an abstract domain (say signs of integers). 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:

1. a semantics of the language is defined, called the standard semantics;

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

2. the standard semantics is transformed into a collecting semantics (or concrete semantics);

3. the collecting semantics is abstracted into an abstract semantics;

4. the abstract semantics (or a subset of it) is computed.

The second step is needed in general because abstract interpretation aims at giving some information about the results of a computation for a set of input values, not a single input value.

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, where constraint stores are 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 [48]. See also [15, 15, 18, 20, 30, 29] for applications of static analysis to CLP.

Normalized Programs A $\texttt{CLP}(\Re_{Lin})$ 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 $\texttt{p(t}_1\texttt{,...,t}_n\texttt{)}$ where $\texttt{t}_1\texttt{,...,t}_n$ 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 $\texttt{CLP}(\Re_{Lin})$ can be specified as follows:

Definition 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 $\texttt{CLP}(\Re_{Lin})$ 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. A linear constraint has the form $c_0\ \delta\ c_1 x_{i_1} + \ldots + c_m x_{i_m}$ where $\delta \in \{=, <, \leq, >, \geq, \neq\}$ and each c_i is a rational number. It is not a difficult matter to transform any given $\texttt{CLP}(\Re_{Lin})$ program into an equivalent normalized program by simple rewriting rules. The use of normalized programs was first proposed for logic programs in [4].

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. In the following, we denote multiset union by $\underline{\cup}$, 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 $\texttt{Var}(\theta)$ to denote the set of variables with non-zero coefficients in θ. We also denote by $\aleph$ the set $\{i \mid x_i \in D\}$.

Concrete Operations The concrete semantics of CLP programs (and, of course, of LP 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. $\texttt{UNION}(\Theta_1, \ldots, \Theta_m)$ 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

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

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. It is specified as follows.

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

$$\texttt{AI_ADD}(\lambda, \Theta) = \{\theta \ \underline{\cup} \ \{\lambda\} \mid \theta \in \Theta\}.$$

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 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`. This section is organized as follows. 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. Section 4.5 presents a complete worked example of the information collected during program analysis with the domain `LSign`.

4.1 The Domain

As mentioned in the Introduction, the domain was first introduced in [38] and then revisited in [46]. The presentation here is based on [46] which generalizes slightly the domain of [46] and clearly separates multiplicity information from the abstract constraints.[2] 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, [38] 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 $D = \{x_1, \ldots, x_n\}$. Note however, that in our examples, we may use other variable names (from the mortgage example presented earlier).

Definition 2 [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 concretization function $Cc : \texttt{Sign} \rightarrow 2^{\Re}$ is defined as

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

From the definition, it is easy to see that the concretization function is monotone, i.e.

$$s_1 \sqsubseteq s_2 \Rightarrow Cc(s_1) \subseteq Cc(s_2).$$

Definition 3 [Operators] An *operator* is an element of $\texttt{Op} = \{<, \leq, =\}$. Operators are denoted by the letter δ.

Definition 4 [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

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

[2] This was motivated by previous work on sequences [31, 3] which separates properties of the elements of the sequences from properties of the sequence.

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

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

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

Example 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 monotonicity of the concretization function w.r.t. the ordering for abstract constraints follows easily from the definitions.

Lemma 1 If σ_1 and σ_2 are abstract constraints then $\sigma_1 \sqsubseteq \sigma_2 \Rightarrow Cc(\sigma_1) \subseteq Cc(\sigma_2)$.

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

Definition 5 [Multiplicities] A *multiplicity* is an element of $\mathtt{Mult} = \{One, ZeroOrOne, Any\}$. $\mathtt{Mult}$ is ordered by $One \sqsubseteq ZeroOrOne \sqsubseteq Any$. Multiplicities are denoted by the letter μ, possibly subscripted.

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

Definition 6 [Abstract Constraints with Multiplicity] An *abstract constraint with multiplicity* over D is an expression of the form $\langle \sigma, \mu \rangle$, where σ is an abstract constraint over D and μ is a multiplicity. The set of abstract constraints with multiplicity over D is denoted by AM_D and is ordered by

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

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

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

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

Example 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., $\{3 = \mathtt{P} + \mathtt{R}\}$. $\langle 0 \leq \oplus\mathtt{P} + \ominus\mathtt{R}, \mathtt{Any} \rangle$ represents multisets of any size, e.g., $\emptyset$, $\{0 \leq 3\mathtt{P} - \mathtt{R}\}$ and $\{0 \leq 3\mathtt{P} - \mathtt{R}, 0 \leq 2\mathtt{P} - 3\mathtt{R}\}$.

[3] [38] 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 [38].

It follows easily from the definition of abstract constraints with multiplicity that the concretization is monotone w.r.t. the ordering.

Lemma 2 If γ_1 and γ_2 are abstract constraints with multiplicity then $\gamma_1 \sqsubseteq \gamma_2 \Rightarrow Cc(\gamma_1) \subseteq Cc(\gamma_2)$.

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

Definition 7 [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 $Cc : AS_D \rightarrow CS_D$ is defined in two stages. The first captures the syntactic form of the abstract store.

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

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

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 8 [Definite, Possible, and Indefinite Constraints] Let γ be an abstract constraint with multiplicity. γ is a *definite* abstract constraint iff $mult(\gamma) = One$. γ is a *possible* abstract constraint iff $mult(\gamma) = ZeroOrOne$. It is an *indefinite* abstract constraint otherwise. The *definite portion* of β, denoted by $\mathtt{Def}(\beta)$, 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 9 [Ordering Function] Let β_1 and β_2 be two abstract stores. An *ordering function* of β_1 to β_2 is a function $f : \beta_1 \rightarrow \beta_2$ satisfying

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

Definition 10 [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 simply 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[4].

Example 4 Consider the following abstract constraints with multiplicity.

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

$\{\gamma_1, \gamma_2\} \sqsubseteq \{\gamma_4, \gamma_5\}$ and the ordering function is defined by $f(\gamma_1) = \gamma_4$ and $f(\gamma_2) = \gamma_5$.
$\{\gamma_1, \gamma_2, \gamma_3\} \sqsubseteq \{\gamma_4, \gamma_5\}$ and the function is defined by $f(\gamma_1) = \gamma_4$, $f(\gamma_2) = \gamma_5$, and $f(\gamma_3) = \gamma_5$.
$\{\gamma_3, \gamma_4\} \not\sqsubseteq \{\gamma_1, \gamma_5, \gamma_6\}$, 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 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 11 [Lifting Function] Let β be an abstract store and θ a concrete constraint store. A *lifting function* of θ to β is a function $f : \theta \to \beta$ satisfying

[4]Notice that the ordering on abstract stores defined in [38], i.e.,

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

does not capture the intended meaning, since it would conclude that the set $\beta_1 = \{\oplus = \oplus x_1 + \oplus x_2, \oplus = \oplus x_1 + \ominus x_2\}$ is smaller than $\beta_2 = \{\oplus = \oplus x_1 + \top x_2\}$. The intended meaning of [38] is that β_1 represents exactly two constraints while β_2 represents exactly one constraint. We observed this problem when trying to define an upper bound operation for `LSign` which was not given in [38]. Obviously, if two distinct clauses return $\{\oplus = \oplus x_1 + \oplus x_2\}$ and $\{\oplus = \oplus x_1 + \ominus x_2\}$, we were tempted to return $\{\oplus = \oplus x_1 + \top x_2\}$ as an upper bound but noticed that $\beta_1 = \{\oplus = \oplus x_1 + \oplus x_2, \oplus = \oplus x_1 + \ominus x_2\}$ was smaller w.r.t. the defined ordering. This ordering problem obviously made it impossible to prove the correctness of the abstract operations of `LSign`.

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

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

Lemma 3 [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 $|\beta| = n+1$ and let β be $\{\gamma\} \cup \beta'$, where $\gamma \notin \beta'$ and $|\beta'| = n$. By Definition 7,

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

($\Rightarrow$) Let $\theta \in Cc_i(\beta)$. Hence $\theta = \theta_1 \underline{\cup} \theta'$, 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 an 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 11 and $\theta_1 \in Cc(\gamma)$ by Definition 6. Let $\theta = \theta_1 \underline{\cup} \theta'$ 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 7, $\theta_1 \underline{\cup} \theta' \in Cc_i(\{\gamma\} \cup \beta')$.

□

Example 5 Consider the following abstract constraints with multiplicity

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

and concrete constraints

$$\begin{array}{rcl} \lambda_1 & = & 2 = 2\mathtt{P} + 3\mathtt{R} \\ \lambda_2 & = & 3 = -2\mathtt{P} + 2\mathtt{R} \\ \lambda_3 & = & 1 = -3\mathtt{P} + 2\mathtt{R} \\ \lambda_4 & = & 4 = 4\mathtt{P} + 3\mathtt{R} \end{array}$$

$\{\lambda_1, \lambda_2\} \in Cc_i(\{\gamma_2, \gamma_3\})$ and the lifting function is defined by $f(\lambda_1) = \gamma_2$ and $f(\lambda_2) = \gamma_3$.
$\{\lambda_1, \lambda_2, \lambda_3\} \in Cc_i(\{\gamma_2, \gamma_3\})$ and the function is defined by $f(\lambda_1) = \gamma_2$, $f(\lambda_2) = \gamma_3$, and $f(\lambda_3) = \gamma_3$.
$\{\lambda_3, \lambda_4\} \notin Cc_i(\{\gamma_1, \gamma_2, \gamma_4\})$, 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 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 3, there exists a lifting function f of θ to β_1. By Definition 10, there exists an ordering function g of β_1 to β_2. Consider the function $g \circ f$. The proof consists of showing that $g \circ f$ is a lifting function of θ to β_2.
(ii) Now 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.

4.2.1 Ordering of Constraint Stores

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

Definition 13 [Maximum Weighted Bipartite Graph Matching Problem] Let $G = (V, E)$ be a weighted bipartite graph. A matching M on G is a set of edges no two of which have a common vertex. The weight of M is the sum of its edge weights. The *maximum weighted bipartite graph matching problem* is that of finding a matching of maximum weight.

The key ideas behind the reduction are as follows. The first set of vertices corresponds to the constraints of β_1. The second set of vertices contains a vertex for each constraint in $\texttt{Def}(\beta_2)$, a vertex for each constraint in $\texttt{Pos}(\beta_2)$, and $|\beta_1|$ vertices for each constraint in $\texttt{Indef}(\beta_2)$, since these constraints can be used several times and the matching problem requires that each vertex appears 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 14 Let β_1 and β_2 be two abstract stores. The *matching graph* of β_1 to β_2 is a (weighted bipartite) graph $G = (V, E)$ such that

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

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

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

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

Sketch of Proof: ($\Rightarrow$) Let $\beta_1 \sqsubseteq \beta_2$. By Definition 10, 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,

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

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

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

We define f by $f(\gamma_1) = \gamma_2$ if $(\gamma_1, \gamma_2) \in M_2$, or $(\gamma_1, \gamma_2) \in M_3$, or $(\gamma_1, {\gamma_2}^{\gamma_1}) \in M_4$ and $f(\gamma_1) = \textit{undefined}$ 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. □

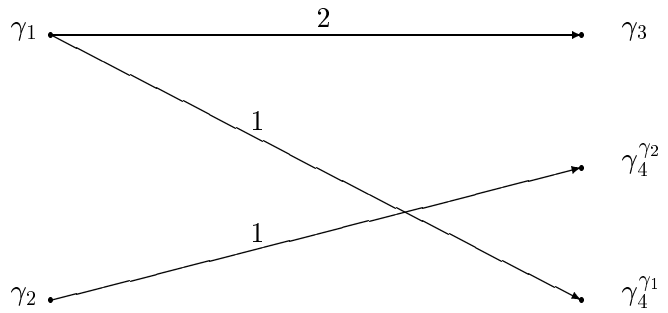
Theorem 3 The implementation of ordering satisfies its definition.

Proof: This follows from Lemma 4 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 6 Consider the abstract stores $\beta_1 = \{\gamma_1, \gamma_2\}$ and $\beta_2 = \{\gamma_3, \gamma_4\}$, where

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

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 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 [17] which proved that the weighted bipartite matching problem can be solved in time $O(|V|^2 \log |V| + |V||E|)$ i.e $O(|\beta_1|^2 |\beta_2|^2 \log |\beta_1||\beta_2|)$ and the definition of the matching graph. □

4.2.2 Addition of a Constraint to a Constraint Store

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

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

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

Informally speaking, the operation adds γ to β if γ is not in β. Otherwise, the multiplicity needs to be adjusted to take into account the new constraint. This is done by setting the multiplicity to *Any*. Although the operation is very simple, the proof of its consistency is non-trivial and indicates why it is convenient to consider multisets (and not sets) of constraints in the concrete semantics.[5]

Example 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 gives the abstract store

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

Adding the abstraction of $0 \leq 3\mathtt{R} + 4\mathtt{B}$, i.e. $\langle 0 \leq \oplus\mathtt{R} + \oplus\mathtt{B}, \mathit{One}\rangle$, 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 5 Let β be an abstract store and γ be an abstract constraint with multiplicity.
(i) If $\theta_1 \in \mathit{Cc}_i(\beta)$ and $\theta_2 \in \mathit{Cc}(\gamma)$, then $\theta_1 \underline{\cup}\, \theta_2 \in \mathit{Cc}_i(\beta \uplus \gamma)$.
(ii) If $\theta_1 \in \mathit{Cc}(\beta)$ and $\theta_2 \in \mathit{Cc}(\gamma)$, then $\theta_1 \underline{\cup}\, \theta_2 \in \mathit{Cc}(\beta \uplus \gamma)$.

Sketch of Proof: Let $\gamma = \langle \sigma, \mu\rangle$. If $\theta_1 \in \mathit{Cc}_i(\beta)$, then there exists a lifting function f of θ_1 to β by Lemma 3. Consider the function $f' : \theta_1 \underline{\cup}\, \theta_2 \to \beta \uplus \gamma$ given by

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

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

Example 8 From the definition of the concretization,

$$\{1 = \mathtt{P} + \mathtt{R}\} \in \mathit{Cc}_i(\{\langle \oplus = \oplus\mathtt{P} + \oplus\mathtt{R}, \mathtt{One}\rangle\}) \ \wedge\ 5 = 2\mathtt{P} + 3\mathtt{R} \in \mathit{Cc}(\langle \oplus = \oplus\mathtt{P} + \oplus\mathtt{R}, \mathtt{One}\rangle).$$

By Definition 15 and Theorem 5 (i), we have that

$$\{1 = \mathtt{P} + \mathtt{R}, 5 = 2\mathtt{P} + 3\mathtt{R}\} \in \mathit{Cc}_i(\{\langle \oplus = \oplus\mathtt{P} + \oplus\mathtt{R}, \mathtt{One}\rangle,\ \langle \oplus = \oplus\mathtt{P} + \oplus\mathtt{R}, \mathtt{Any}\rangle\})..$$

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

Definition 16 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 5 Let β and β' be abstract stores.
(i) If $\theta \in \mathit{Cc}_i(\beta)$ and $\theta' \in \mathit{Cc}_i(\beta')$ then $\theta \underline{\cup}\, \theta' \in \mathit{Cc}_i(\beta \biguplus \beta')$.
(ii) If $\theta \in \mathit{Cc}(\beta)$ and $\theta' \in \mathit{Cc}(\beta')$ then $\theta \underline{\cup}\, \theta' \in \mathit{Cc}(\beta \biguplus \beta')$.

[5] It is in fact possible to be more precise in certain cases by using multiplicity *ZeroOrOne* when $\langle \sigma, \mathit{ZeroOrOne}\rangle$ is not in β. We gave the above definition for simplicity.

Example 9 The following example is based on the mortgage program `mg/4`. 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 5 that $\theta \underline{\cup} \theta' \in Cc(\beta \biguplus \beta')$.

4.2.3 Upper Bound of Constraint Stores

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

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

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

Definition 18 [Upper Bound on Multiplicities] The upper bound operation on multiplicities $\sqcup$: `Mult` $\times$ `Mult` $\to$ `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

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

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.

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.

$$
\begin{array}{lll}
1] & \texttt{UNION}(\emptyset, \emptyset) = \emptyset & \\
2\ i] & \texttt{UNION}(\{\langle \sigma, One\rangle\} \cup \beta_1', \beta_2) = \texttt{UNION}(\beta_1', \beta_2) \ \uplus\ \langle \sigma, ZeroOrOne\rangle & \text{if } \langle \sigma, One\rangle \notin \beta_1' \\
ii] & \texttt{UNION}(\beta_1, \{\langle \sigma, One\rangle\} \cup \beta_2') = \texttt{UNION}(\beta_1, \beta_2') \ \uplus\ \langle \sigma, ZeroOrOne\rangle & \text{if } \langle \sigma, One\rangle \notin \beta_2' \\
3\ i] & \texttt{UNION}(\{\langle \sigma, \mu\rangle\} \cup \beta_1', \beta_2) = \texttt{UNION}(\beta_1', \beta_2) \ \uplus\ \langle \sigma, \mu\rangle & \text{if } \langle \sigma, \mu\rangle \notin \beta_1' \wedge \mu \neq One \\
ii] & \texttt{UNION}(\beta_1, \{\langle \sigma, \mu\rangle\} \cup \beta_2') = \texttt{UNION}(\beta_1, \beta_2') \ \uplus\ \langle \sigma, \mu\rangle & \text{if } \langle \sigma, \mu\rangle \notin \beta_2' \wedge \mu \neq One
\end{array}
$$

$$
\begin{array}{ll}
4] & \texttt{UNION}(\{\langle s_0 \ \delta\ \sum_{i=1}^{n} s_i x_i, \mu\rangle\} \cup \beta_1', \{\langle s_0' \ \delta\ \sum_{i=1}^{n} s_i' x_i, \mu'\rangle\} \cup \beta_2') = \\
& \quad \texttt{UNION}(\beta_1', \beta_2') \ \uplus\ \langle (s_0 \sqcup s_0') \ \delta\ \sum_{i=1}^{n} (s_i \sqcup s_i') x_i, \mu \sqcup \mu'\rangle \\
& \quad \text{if } \langle s_0 \ \delta\ \sum_{i=1}^{n} s_i x_i, \mu\rangle \notin \beta_1' \text{ and } \langle s_0' \ \delta\ \sum_{i=1}^{n} s_i' x_i, \mu'\rangle \notin \beta_2'
\end{array}
$$

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

4.2.4 Projection of a Constraint Store

We now turn to the projection operation. Figure 1 describes a simple projection algorithm based on the traditional Gaussian and Fourier eliminations which are standard in this area. Figure 2 presents the abstract algorithm. The algorithms are close to those in [38] 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

```
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 = {σ'}⊔θ' in
    ⟨θ1^0, θ1^+, θ1^-⟩ := Csplit({σ'}, 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;
}
```

⋈	<	≤	=
<	<	<	<
≤	<	≤	≤
=	<	≤	=

```
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 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 2: Abstract Projection Algorithms

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 presentation is slightly more complicated than required in order to make the abstract version simpler.

The abstract version mimics almost line by line the concrete version showing the benefits of using operations `Csplit` and `Asplit`. The only place where we deal with $\top$ is precisely in `Asplit_top`. This keeps the rest of the abstract Fourier algorithm as close as possible to the concrete version. 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 $\Re$. More formally,

$$\begin{gathered} c_1 \in Cc(s_1) \wedge c_2 \in Cc(s_2) \Rightarrow c_1 + c_2 \in Cc(s_1 + s_2) \\ c_1 \in Cc(s_1) \wedge c_2 \in Cc(s_2) \Rightarrow c_1 - c_2 \in Cc(s_1 - s_2) \\ c_1 \in Cc(s_1) \wedge c_2 \in Cc(s_2) \Rightarrow c_1 \times c_2 \in Cc(s_1 \times s_2) \\ c \in Cc(s) \Rightarrow -c \in Cc(-s) \end{gathered}$$

The correctness of the abstract version can be proven by showing the consistency of the various operations. All these (technical) consistency results can be found in [48].

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 [38], 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 11 Consider the mortgage example `mg/4` when called with the top level input pattern $\{\langle \top = \oplus\mathtt{P}, \mathtt{One}\rangle,\ \langle \top = \oplus\mathtt{R}, \mathtt{One}\rangle\}$. For the first execution of the second clause, the abstract store describing the computation at the point just before the recursive call to `mg` is

$$\begin{gathered} \{\langle \top = \oplus\mathtt{P}, \mathtt{One}\rangle,\ \langle \top = \oplus\mathtt{R}, \mathtt{One}\rangle,\ \langle 0 < \oplus\mathtt{T}, \mathtt{One}\rangle, \\ \langle 0 \leq \oplus\mathtt{P}, \mathtt{One}\rangle,\ \langle 0 = \oplus\mathtt{P} + \ominus\mathtt{R} + \ominus\mathtt{P1}, \mathtt{One}\rangle,\ \langle \oplus = \oplus\mathtt{T} + \ominus\mathtt{T1}, \mathtt{One}\rangle\} \end{gathered}$$

Projecting `T` from the above store corresponds to computing $\mathtt{Agauss}(\beta,\ \oplus = \oplus\mathtt{T} + \ominus\mathtt{T1},\ \mathtt{T})$ where β is

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

The resulting abstract store is

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

Projecting `T` from the above store corresponds to computing $\mathtt{Agauss}(\beta',\ 0 = \oplus\mathtt{P} + \ominus\mathtt{R} + \ominus\mathtt{P1},\ \mathtt{P})$ where β' is

$$\{\langle \top = \oplus\mathtt{P}, \mathtt{One}\rangle,\ \langle \top = \oplus\mathtt{R}, \mathtt{One}\rangle,\ \langle \ominus < \oplus\mathtt{T1}, \mathtt{One}\rangle, \langle 0 \leq \oplus\mathtt{P}, \mathtt{One}\rangle\}$$

The resulting abstract store (which represents the input to the recursive call `mg(P1,T1,R,B)`) is

$$\{\langle \top = \oplus \mathtt{R} + \oplus \mathtt{P1}, \mathtt{One}\rangle,\ \langle \top = \oplus \mathtt{R}, \mathtt{One}\rangle,\ \langle \ominus < \oplus \mathtt{T1}, \mathtt{One}\rangle, \langle 0 \leq \oplus \mathtt{R} + \oplus \mathtt{P1}, \mathtt{One}\rangle\}$$

Note that the equation $\top = \oplus \mathtt{P}$ could have been used instead to eliminate `P` from β', giving a different result.

Operation `Asplit_top` 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 $\ominus$. `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. The concrete version is made more complicated than necessary in order to simplify the presentation of the abstract version. 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 [38]. This is achieved by testing the multiplicity of the constraint.

Example 12 Let $\gamma_1 = \langle 0 \leq \oplus \mathtt{R} + \oplus \mathtt{B}, \mathtt{One}\rangle$, $\gamma_1 = \langle 0 = \oplus \mathtt{P} + \top \mathtt{R} + \ominus \mathtt{B}, \mathtt{One}\rangle$ 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.

Theorem 7 [Project] Let $v \in \aleph$, θ be a store and β be an abstract store.

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

Corollary 1 [Project Set] Let $V \in 2^{\aleph}$, 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 [38]. The first improvement is on the accuracy of abstract Fourier elimination. In [38], 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 $\{\langle \top = \top x + \oplus y, ZeroOrOne\rangle\}$ leads to the store $\{\langle \top = \oplus y, ZeroOrOne\rangle, \langle \top = \oplus y, Any\rangle\}$ using the algorithm of [38]. 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 $\{\langle \top = \oplus y, ZeroOrOne\rangle\}$, 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 [38], projecting x from $\{\langle \oplus = \oplus x, One\rangle, \langle \top = \top x + \oplus y, One\rangle\}$ involves considering the cases 0, $\oplus$ and $\ominus$ for the $\top$ coefficient of x in the second constraint, leading to $\{\langle \top = \oplus y, ZeroOrOne\rangle, \langle \top = \oplus y, Any\rangle\}$ which may or may not be satisfiable. However, by directly substituting $\langle \oplus = \oplus x, One\rangle$ into the second constraint, we get a simpler abstract store $\{\langle \top = \oplus y, One\rangle\}$ 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 [48] 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. It starts by discussing satisfiability and unsatisfiability of constraint stores and then move to the operations itself.

4.3.1 Satisfiability of Constraint Stores

Consider first the problem of determining whether a constraint store is satisfiable. In the concrete domain, this problem can be formalized by a function $\texttt{Cis_sat} : CS_D \rightarrow$ Boolean which takes a constraint store θ and returns a Boolean value which is true if θ is satisfiable

```
Cis_sat(θ) {
θ_p = Cproject_set(θ, ℵ);
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}^n 0x_i ∧ c_0 = 0:
    case λ is c_0 ≤ Σ_{i=1}^n 0x_i ∧ c_0 ≤ 0:
        return Cis_triv_sat(θ');
    otherwise:
        return false;
}
```

```
Ais_sat(β) {
β_p = Aproject_set(β, ℵ);
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}^n 0x_i ∧ s_0 = 0:
    case σ is s_0 ≤ Σ_{i=1}^n 0x_i ∧ (s_0 = 0 ∨ s_0 = ⊖):
        return Ais_triv_sat(β');
    otherwise:
        return false;
}
```

Figure 3: Satisfiability Algorithms

and false otherwise. An implementation of `Cis_sat` is given in Figure 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 3, makes a conservative approximation to `Cis_sat` that satisfies the following specification.

Specification 6 `Ais_sat` : $AS_D \rightarrow$ Boolean should satisfy the following consistency condition. $\forall \theta \in CS_D, \forall \beta \in AS_D$:

$$\theta \in Cc(\beta) \Rightarrow (\texttt{Ais_sat}(\beta) \Rightarrow \texttt{Cis_sat}(\theta))$$

Example 13 Projecting all the variables from $\{0 \leq \texttt{R} + \texttt{B},\ 0 = \texttt{B},\ 3 = \texttt{R}\}$ gives the constraint store $\{-3 \leq 0\}$, indicating that the original constraint store is satisfiable.

Example 14 Projecting all the variables from $\{\langle\ 0 \leq \oplus\texttt{R} + \oplus\texttt{B},\ \texttt{One}\ \rangle,\ \langle\ 0 = \oplus\texttt{B},\ \texttt{One}\ \rangle,\ \langle\ \oplus = \oplus\texttt{R},\ \texttt{One}\ \rangle\}$ gives the abstract store $\{\langle\ \ominus \leq 0,\ \texttt{One}\ \rangle\}$, indicating that all the constraint stores in its concretization are satisfiable.

Example 15 Projecting all the variables from $\{\langle\ \top \leq \oplus\texttt{R} + \oplus\texttt{B},\ \texttt{One}\ \rangle,\ \langle\ 0 = \oplus\texttt{B},\ \texttt{One}\ \rangle,\ \langle\ \oplus = \oplus\texttt{R},\ \texttt{One}\ \rangle\}$ gives the abstract store $\{\langle\ \top \leq 0,\ \texttt{One}\ \rangle\}$, indicating that some of the constraint stores in its concretization *may not* be satisfiable.

The following theorem proves the correctness of `Ais_sat`.

Theorem 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_unsat(θ) {
θ_p = Cproject_set(θ, ℵ);
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}^n 0x_i ∧ c_0 ≠ 0:
    case λ is c_0 ≤ Σ_{i=1}^n 0x_i ∧ c_0 > 0:
        return true;
    otherwise:
        return Cis_triv_unsat(θ');
}


Ais_unsat(β) {
β_p = Aproject_set(β, ℵ);
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: Unsatisfiability Algorithms

4.3.2 Unsatisfiability of Constraint Stores

Consider now the problem of determining whether a constraint store is unsatisfiable. In the concrete domain, this problem can be formalized by the function `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`. In order to simplify the presentation of the abstract algorithm, it is useful to give the definition of `Cis_unsat` explicitly in Figure 4. It consists of the well known technique of 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, makes a conservative approximation to `Cis_unsat` that satisfies the following specification.

Specification 7 `Ais_unsat` : $AS_D \rightarrow$ Boolean should satisfy the following consistency condition. $\forall \theta \in CS_D, \forall \beta \in AS_D$:

$$\theta \in Cc(\beta) \Rightarrow (\texttt{Ais_unsat}(\beta) \Rightarrow \texttt{Cis_unsat}(\theta))$$

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

Example 16 Projecting all the variables from $\{0 \leq \texttt{R} + \texttt{B},\ 0 = \texttt{B},\ -3 = \texttt{R}\}$ gives the constraint store $\{3 \leq 0\}$, indicating that the original constraint store is unsatisfiable.

Example 17 Projecting all the variables from $\{\langle\ 0 \leq \oplus\texttt{R} + \oplus\texttt{B},\ \texttt{One}\ \rangle,\ \langle\ 0 = \oplus\texttt{B},\ \texttt{One}\ \rangle,\ \langle\ \ominus = \oplus\texttt{R},\ \texttt{One}\ \rangle\}$ gives the abstract store $\{\langle\ \oplus \leq 0,\ \texttt{One}\ \rangle\}$, indicating that all the constraint stores in its concretization are unsatisfiable.

Example 18 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.

The following theorem proves the correctness of `Ais_unsat`.

Theorem 9 [Is Unsatisfiable] Let θ be a store and β be an abstract store.

$$\theta \in Cc(\beta) \Rightarrow (\mathtt{Ais_unsat}(\beta) \Rightarrow \mathtt{Cis_unsat}(\theta))$$

4.3.3 Conditional Satisfiability of Constraint Stores

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

$$\begin{aligned}\mathtt{Cis_cond_sat}(\theta_1, \theta_2) &= \mathtt{Cis_sat}(\theta_1) \Rightarrow \mathtt{Cis_sat}(\theta_1 \underline{\cup} \theta_2)\\ &= \mathtt{Cis_unsat}(\theta_1)\ \vee\ \mathtt{Cis_sat}(\theta_1 \underline{\cup} \theta_2).\end{aligned}$$

Its abstract counterpart `Ais_cond_sat` is specified as follows.

Specification 8 $\mathtt{Ais_cond_sat} : AS_D \times AS_D \rightarrow$ Boolean should satisfy the following consistency condition. $\forall\theta_1, \theta_2 \in CS_D, \forall\beta_1, \beta_2 \in AS_D$:

$$\theta_1 \in Cc(\beta_1)\ \wedge\ \theta_2 \in Cc(\beta_2) \Rightarrow (\mathtt{Ais_cond_sat}(\beta_1, \beta_2) \Rightarrow \mathtt{Cis_cond_sat}(\theta_1, \theta_2))$$

The need for such an operation arises when it is necessary to verify (in a conservative manner) 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 $\mathtt{CLP}(\Re_{Lin})$ programs. While the definition of `Cis_cond_sat` is straightforward, it does not lend itself to a straightforward abstraction. The obvious implementation of `Ais_cond_sat` is

$$\mathtt{Ais_cond_sat}(\beta_1, \beta_2) = \mathtt{Ais_unsat}(\beta_1)\ \vee\ \mathtt{Ais_sat}(\beta_1 \uplus \beta_2).$$

The correctness of this implementation w.r.t. the specification is an easy consequence of the correctness of `Ais_sat`, `Ais_unsat` and the $\uplus$ operator. However, as we shall see below, this implementation may not be sufficiently accurate. In the rest of this section, we examine why the obvious implementation of `Ais_cond_sat` is not sufficiently accurate, and how it may be improved.

The basic source of inaccuracy in the abstraction of `Cis_cond_sat` is the fact that the satisfiability of $\theta_1 \underline{\cup} \theta_2$ only needs to be checked if θ_1 is satisfiable. In other words, if θ_1 is known to be unsatisfiable, then there is no need to check the satisfiability of $\theta_1 \underline{\cup} \theta_2$. In the concrete domain, this poses no problems as `Cis_sat` and `Cis_unsat` are logical negations of one another. However, in the abstract domain, the fact that `Ais_unsat` and `Ais_sat` are not logical negations of each other may lead to inaccuracy. Intuitively, it is necessary to check the satisfiability of $\beta_1 \uplus \beta_2$ only for the stores in the concretization of

β_1 that are satisfiable. The approach taken here is to remove as many unsatisfiable stores as possible from the concretization of β_1, before computing $\texttt{Ais_sat}(\beta_1 \uplus \beta_2)$. Intuitively, this approach leads to a more precise approximation of $\texttt{Cis_cond_sat}(\theta_1, \theta_2)$ because we only need to check the satisfiability of $\theta_1 \underline{\cup} \theta_2$ if θ_1 is known to be satisfiable. There are four distinct, but inter-related, ways to remove sources of unsatisfiability from β_1. We now examine each of these in detail, with the aid of a motivating example.

Removing Ground Constraints In the concrete domain, the transformation `Cred_gnd` (Figure 5) removes all ground constraints (i.e. constraints with zero coefficients for all variables) from a constraint store. `Cred_gnd` only removes sources of trivial unsatisfiability from constraint stores and it is an equivalence transformation for satisfiable constraint stores.

Proposition 1 Let θ be a constraint store. Then

$$\theta \text{ satisfiable } \Rightarrow (\theta \leftrightarrow \texttt{Cred_gnd}(\theta)).$$

While `Cred_gnd` is not a very useful transformation for constraint stores, the following example indicates why its abstraction `Ared_gnd` (also given in Figure 5) is important to improve the accuracy of conditional satisfiability in the abstract domain.

Example 19 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 $\texttt{Cis_sat}(\theta_1 \underline{\cup} \theta_2)$ 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. It can be observed however that 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

$$\texttt{Ais_cond_sat}(\beta_1, \beta_2) = \texttt{Ais_unsat}(\beta_1) \ \vee \ \texttt{Ais_sat}(\texttt{Ared_gnd}(\beta_1) \uplus \beta_2).$$

Eliminating Equations In the concrete domain, the transformation `Cred_eqn` (Figure 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 2 Let $v \in \aleph, V \in 2^{\aleph}$ 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 5) is important to improve the accuracy of conditional satisfiability in the abstract domain.

Example 20 Let $\beta_1 = \{\langle \oplus = \oplus X, \texttt{One}\rangle, \langle \top < \oplus X, \texttt{One}\rangle\}$ and $\beta_2 = \{\langle 0 = \oplus X + \oplus Y, \texttt{One}\rangle\}$. 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 $\texttt{Cis_sat}(\theta_1 \cup \theta_2)$ 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 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 $\texttt{Ais_sat}(\beta_1' \uplus \beta_2)$ 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 21 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 $\texttt{Cis_sat}(\theta_1 \cup \theta_2)$ 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 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 $\texttt{Ais_sat}(\beta_1' \uplus \beta_2)$ 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 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 can all be put together in a transformation called the definite satisfiability *reduction* (Figure 5). In the concrete domain, the operation `Creduce` : $CS_D \rightarrow CS_D$ consists of the composition of `Cred_eqn` and `Cred_gnd`. In the abstract domain, the operation `Areduce` : $AS_D \rightarrow AS_D$ consists of the composition of `Ared_top`, `Ared_eqn` and `Ared_gnd`. The definition of `Areduce` is such that, while it retains all the satisfiable stores in the concretization, it removes as many unsatisfiable stores as possible. Note however that while the definition of `Areduce` is such that the concretization of $\texttt{AReduce}(\beta)$ contain all the satisfiable stores from the concretization of β, it does not preclude the introduction of stores into $Cc(\texttt{Areduce}(\beta))$ that are not in $Cc(\beta)$. The following theorem states this more formally.

Theorem 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:

$$\texttt{Ais_cond_sat}(\beta_1, \beta_2) = \texttt{Ais_unsat}(\beta_1) \ \vee \ \texttt{Ais_sat}(\texttt{Areduce}(\beta_1) \biguplus \beta_2).$$

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

Example 22 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 $\texttt{Cis_sat}(\theta_1 \underline{\cup} \theta_2)$. We expect therefore that $\texttt{Ais_cond_sat}(\beta_1, \beta_2)$ returns true. However we see that

$$\begin{aligned} \texttt{Ais_cond_sat}(\beta_1, \beta_2) &= \texttt{Ais_unsat}(\beta_1) \ \vee \ \texttt{Ais_sat}(\texttt{Areduce}(\beta_1) \biguplus \beta_2) \\ &= \text{false} \ \vee \ \texttt{Ais_triv_sat}(\{\langle \top < 0, \texttt{ZeroOrOne}\rangle\}) \\ &= \text{false} \end{aligned}$$

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.

```
Creduce(θ) {
return Cred_gnd(Cred_eqn(θ));
}

Cred_gnd(θ) {
if θ = ∅ return ∅;
else let θ = {λ} ⊔ θ' in
    case λ is c_0 δ Σ_{i=1}^{n} 0x_i:
        return Cred_gnd(θ');
    otherwise:
        return Cred_gnd(θ') ⊔ {λ};
}

Cred_eqn(θ) {
return Cred_eqn_set(θ, ℵ);
}

Cred_eqn_set(θ, V) {
if V = ∅ return θ;
else let V = {v} ∪ V' ∧ v ∉ V' in
    θ_v := Cred_eqn_step(θ, v);
    return Cred_eqn_set(θ_v, V');
}

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

```
Areduce(β) {
return Ared_gnd(Ared_eqn(Ared_top(β)));
}

Ared_gnd(β) {
if β = ∅ return ∅;
else let β = {γ} ∪ β' ∧ γ = ⟨σ, μ⟩ ∉ β' in
    case σ is s_0 δ Σ_{i=1}^{n} 0x_i:
        return Ared_gnd(β');
    otherwise:
        return Ared_gnd(β') ⊎ γ;
}

Ared_eqn(β) {
return Ared_eqn_set(β, ℵ);
}

Ared_eqn_set(β, V) {
if V = ∅ return β;
else let V = {v} ∪ V' ∧ v ∉ V' in
    β_v := Ared_eqn_step(β, v);
    return Ared_eqn_set(β_v, V');
}

Ared_eqn_step(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) ⊎ γ;
    case μ = One ∧ op(σ) is '=' ∧ σ[v] = ⊖:
        return Agauss(β',Aneg(σ),v) ⊎ γ;
    otherwise:
        return β;
}
```

```
Ared_top(β) {
if β = ∅ return ∅;
else if |β| = 1 return β;
else let β = {γ_1, γ_2} ∪ β' ∧ γ_1 = ⟨σ_1, μ_1⟩ ∉ β' ∧ γ_2 = ⟨σ_2, μ_2⟩ ∉ β' in
    case μ_1 = One ∧ (σ_1 is ⊕ ≤ ⊕x_v ∨ σ_1 is 0 < ⊕x_v ∨ σ_1 is ⊕ < ⊕x_v)
        ∧(σ_2 is ⊤ = ⊕x_v ∨ σ_2 is ⊤ = ⊖x_v):
        return Ared_top(β' ⊎ γ_1) ⊎ ⟨⊕ = ⊕x_v, μ_2⟩;
    case μ_1 = One ∧ (σ_1 is ⊕ ≤ ⊖x_v ∨ σ_1 is 0 < ⊖x_v ∨ σ_1 is ⊕ < ⊖x_v)
        ∧(σ_2 is ⊤ = ⊕x_v ∨ σ_2 is ⊤ = ⊖x_v):
        return Ared_top(β' ⊎ γ_1) ⊎ ⟨⊖ = ⊕x_v, μ_2⟩;
    otherwise:
        return β;
}
```

Figure 5: Reduction Algorithms

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

$$\begin{array}{l}\texttt{Cis_sat}(\theta_1 \mathbin{\underline{\cup}} \theta_2) \\ = \texttt{Cis_triv_sat}(\texttt{Cproject_set}(\theta_1 \mathbin{\underline{\cup}} \theta_2, \aleph)) \\ = \texttt{Cis_triv_sat}(\texttt{Cproject_set}(\texttt{Cproject_set}(\theta_1, \texttt{Var}(\theta_1) \setminus \texttt{Var}(\theta_2)) \mathbin{\underline{\cup}} \theta_2, \aleph)) \\ = \texttt{Cis_sat}(\texttt{Cproject_set}(\theta_1, \texttt{Var}(\theta_1) \setminus \texttt{Var}(\theta_2)) \mathbin{\underline{\cup}} \theta_2)\end{array}$$

This suggests the following implementation for conditional satisfiability.

Implementation 3 [Is Conditionally Satisfiable] The abstract conditional satisfiability operation $\texttt{Ais_cond_sat} : AS_D \times AS_D \rightarrow$ Boolean is given as

$$\begin{array}{l}\texttt{Ais_cond_sat}(\beta_1, \beta_2) \\ = \texttt{Ais_unsat}(\beta_1) \ \vee \ \texttt{Ais_sat}(\texttt{Areduce}(\texttt{Aproject_set}(\beta_1, \texttt{Var}(\beta_1) \setminus \texttt{Var}(\beta_2))) \uplus \beta_2).\end{array}$$

where $\texttt{Var}(\beta)$ denotes 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 11 [Conditional Satisfiability] Let θ_1, θ_2 be stores and β_1, β_2 be abstract stores. Then

$$\theta_1 \in Cc(\beta_1) \ \wedge \ \theta_2 \in Cc(\beta_2) \Rightarrow (\texttt{Ais_cond_sat}(\beta_1, \beta_2) \Rightarrow \texttt{Cis_cond_sat}(\theta_1, \theta_2))$$

4.3.4 Summary

The naive implementation of $\texttt{Ais_cond_sat}(\beta_1, \beta_2)$ 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 before checking the satisfiability of $\beta_1 \uplus \beta_2$. There are four transformations to remove sources of unsatisfiability. Two of these, i.e., eliminating with equations and projecting irrelevant variables, correspond closely to the concrete domain. They 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.

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 [16]. We briefly introduce the domain $2^{\texttt{LSign}}$ and its abstract operations.

$\alpha \uplus \gamma = \bigcup_{\beta \in \alpha} \beta \uplus \gamma$
$\alpha \biguplus \alpha' = \bigcup_{\beta \in \alpha, \beta' \in \alpha'} \beta \biguplus \beta'$
$\texttt{Aproject}(\alpha, v) = \bigcup_{\beta \in \alpha} \texttt{Aproject}(\beta, v)$
$\texttt{Aproject_set}(\alpha, V) = \bigcup_{\beta \in \alpha} \texttt{Aproject_set}(\beta, V)$
$\texttt{Ais_sat}(\alpha) = \wedge_{\beta \in \alpha} \texttt{Ais_sat}(\beta)$
$\texttt{Ais_unsat}(\alpha) = \wedge_{\beta \in \alpha} \texttt{Ais_unsat}(\beta)$
$\texttt{Ais_cond_sat}(\alpha_1, \alpha_2) = \wedge_{\beta_1 \in \alpha_1, \beta_2 \in \alpha_2} \texttt{Ais_cond_sat}(\beta_1, \beta_2)$
$\texttt{Ais_redundant}(\alpha_r, \alpha) = \wedge_{\beta_r \in \alpha_r, \beta \in \alpha} \texttt{Ais_redundant}(\beta_r, \beta)$
$\texttt{Ais_free}(\alpha, v) = \wedge_{\beta \in \alpha} \texttt{Ais_free}(\beta, v)$

Figure 6: Abstract Operations and Applications for $2^{\texttt{LSign}}$

Definition 19 [Abstract Multistores] An *abstract multistore* over D is a set α of abstract stores. The set of multistores over D is denoted by AMS_D and is ordered by

$$\alpha_1 \sqsubseteq \alpha_2 \Leftrightarrow \forall \beta_1 \in \alpha_1 \; \exists \beta_2 \in \alpha_2 : \beta_1 \sqsubseteq \beta_2$$

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

$$Cc(\alpha) = \bigcup_{\beta \in \alpha} Cc(\beta)$$

Abstract multistores are denoted by the letter α, possibly subscripted.

Example 23 Constraint stores such as $\{0 = \texttt{T}\}$, $\{2 = 3\texttt{T}, 0 < \texttt{T}\}$ and $\{3 = \texttt{T}, 0 < 2\texttt{T}\}$ all belong to the concretization of the abstract multistore $\{\{\langle\ 0 = \oplus\texttt{T},\ \texttt{One}\ \rangle\}, \{\langle\ \oplus = \oplus\texttt{T},\ \texttt{One}\ \rangle, \langle\ 0 < \oplus\texttt{T},\ \texttt{One}\ \rangle\}\}$

It is quite easy to extend the definition of the abstract operations and applications of `LSign` to abstract multistores and to prove their consistency. In particular, the ordering function $\sqsubseteq$: $AMS_D \times AMS_D$ is a pre-order and is monotonic w.r.t. the concretization function. Figure 6 contains the definitions of various algorithms for $2^{\texttt{LSign}}$ in terms of the corresponding algorithms for `LSign`.

The only operation that is not a direct application of the corresponding operation for `LSign` is that for the upper bound operation. Unlike `LSign`, the `UNION` operation in $2^{\texttt{LSign}}$ is not defined as a scheme. Rather, the domain lends itself to a very simple and precise formulation of the upper bound.

Definition 20 [Upper Bound on Abstract Multistores] Let α and α' be abstract multistores.

$$\texttt{UNION}(\alpha, \alpha') = \alpha \cup \alpha'$$

Example 24 The abstract multistore $\{\{\langle\ 0 = \oplus\texttt{T},\ \texttt{One}\ \rangle\}, \{\langle\ \oplus = \oplus\texttt{T},\ \texttt{One}\ \rangle, \langle\ 0 < \oplus\texttt{T},\ \texttt{One}\ \rangle\}\}$ is the upper bound of the abstract multistores $\{\{\langle\ \oplus = \oplus\texttt{T},\ \texttt{One}\ \rangle, \langle\ 0 < \oplus\texttt{T},\ \texttt{One}\ \rangle\}\}$ and $\{\{\langle\ 0 = \oplus\texttt{T},\ \texttt{One}\ \rangle\}\}$.

The following theorem follows easily from the definition of the concretization for abstract multistores.

Theorem 12 Let α_1 and α_2 be abstract multistores and θ be a constraint store.

$$\theta \in Cc(\alpha_1) \vee \theta \in Cc(\alpha_2) \;\Rightarrow\; \theta \in Cc(\texttt{UNION}(\alpha_1, \alpha_2)).$$

We would also like to mention here that $2^{\texttt{LSign}}$ enables the abstraction of disequations in the domain in an accurate manner. This is done by extending the definition of the addition operator as follows.

Definition 21 [Addition of a Disequation] Let α be an abstract multistore and γ be the abstract constraint with multiplicity $\langle s_0 \neq \sum_{i=1}^{n} s_i x_i, \mu\rangle$. Then

$$\alpha \uplus \gamma = (\alpha \uplus \langle s_0 < \sum_{i=1}^{n} s_i x_i, \mu\rangle) \bigcup (\alpha \uplus \langle s_0 < \sum_{i=1}^{n} (-s_i) x_i, \mu\rangle).$$

4.5 Complete Example

We now present a complete example indicating how to compute the output and input descriptions in the $2^{\texttt{LSign}}$ domain, for the mortgage program `mg/4`. First we show the computation of the goal independent (or online) output.

Example 25 [Computation of `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

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

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 gives the following abstract substitution as the updated output of the predicate `mg`.

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

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

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

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

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

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

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

The process can be repeated until it gives the following multistore as the fixpoint for the computation of the output of `mg`.

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

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

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

We now show the computation of the input description, when the top level query is such that both `P` and `R` are fixed.

Example 26 [Computation of `LSign` Input Description for `mg/4`] The abstract substitution describing a top level query such that `P` and `R` are fixed, is given as

$$\{\{\langle \top = \oplus \mathtt{P}, \mathtt{One}\rangle, \langle \top = \oplus \mathtt{R}, \mathtt{One}\rangle\}\}$$

When the second clause is executed with this input, the abstract substitution for the program point just before the recursive call to `mg` is

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

When restricted to the variables of the literal `mg(P1,T1,R,B)`, this gives the abstract substitution

$$\{\{\langle \ominus < \oplus \mathtt{T1}, \mathtt{One}\rangle, \langle \top = \oplus \mathtt{R}, \mathtt{One}\rangle, \langle \top = \oplus \mathtt{P1} + \oplus \mathtt{R}, \mathtt{One}\rangle\}\}$$

which on normalization gives the following input substitution for the recursive call to `mg`

$$\{\{\langle \ominus < \oplus \mathtt{T}, \mathtt{One}\rangle, \langle \top = \oplus \mathtt{R}, \mathtt{One}\rangle, \langle \top = \oplus \mathtt{P} + \oplus \mathtt{R}, \mathtt{One}\rangle\}\}$$

The union of this with the previous input gives the following abstract substitution as the updated input to `mg`

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

Reexecuting the second clause with the updated input leads to the following abstract substitution just before the recursive call to `mg`

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

$$\biguplus$$

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

When restricted to the variables of the literal and normalized, this gives

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

Taking the union with the old input gives the updated input

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

The process can be repeated until it gives the following substitution as the fixpoint for the input of `mg`.

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

which can be simplified to

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

by removing the third store which is subsumed by the fourth store.

4.6 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 $\beta_1 = \{\langle \oplus = \oplus x, \mathit{One}\rangle, \langle 0 < \oplus x, \mathit{One}\rangle\}$. It is easy to see that 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 $\beta_2 = \{\langle \oplus = \oplus x, \mathit{One}\rangle, \langle \oplus < \oplus x, \mathit{One}\rangle\}$. 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 and redundancy. Intuitively, the domain is able to remain accurate in the first example because of the special treatment of 0, while it is inaccurate in the second example because all the positive numbers get grouped into one sign. 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. The domain `LInt` closely resembles the domain `LSign`, however as will be seen, the use of intervals enables it to be more accurate in a number of cases. 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` and their concretization. Section 5.2 briefly presents the operations and applications of the domain. The power domain and the widening are presented in Section 5.3. A complete worked example is presented in Section 5.4. Section 5.5 concludes by discussing how `LInt` addresses the limitations of `LSign`. The proofs of the results in this section can be found in the Appendix.

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, by

$$\forall r \in \Re : \ -\infty < r < +\infty.$$

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 $\mathcal{F}^\infty$. We now define the intervals more formally, as well as what they represent.

Definition 22 [Intervals] An *interval* is an element of

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

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

$$\left\{ \begin{array}{rcl} Cc([a_\ell, a_r]) & = & \{c \mid c \in \Re \ \wedge \ a_\ell \leq c \leq a_r\} \\ Cc((a_\ell, a_r]) & = & \{c \mid c \in \Re \ \wedge \ a_\ell < c \leq a_r\} \\ Cc([a_\ell, a_r)) & = & \{c \mid c \in \Re \ \wedge \ a_\ell \leq c < a_r\} \\ Cc((a_\ell, a_r)) & = & \{c \mid c \in \Re \ \wedge \ a_\ell < c < a_r\} \end{array} \right.$$

Intervals are denoted by the letter s, possible subscripted.

We assume the existence of functions $\texttt{left} : \texttt{Intv} \to \mathcal{F}^\infty$ and $\texttt{right} : \texttt{Intv} \to \mathcal{F}^\infty$ 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 $\texttt{increase} : \texttt{Intv} \times \texttt{Intv}$ and $\texttt{decrease} : \texttt{Intv} \times \texttt{Intv}$ as follows:

$$\texttt{increase}(s_1, s_2) \Leftrightarrow \exists c_1 \in Cc(s_1) \ \forall c_2 \in Cc(s_2) : c_1 > c_2$$

$$\texttt{decrease}(s_1, s_2) \Leftrightarrow \exists c_1 \in Cc(s_1) \ \forall c_2 \in Cc(s_2) : c_1 < c_2$$

The ordering $\sqsubseteq : \texttt{Intv} \times \texttt{Intv}$ 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 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 $\Re^\infty$. More formally,

$$\begin{gathered}
c_1 \in Cc(s_1) \wedge c_2 \in Cc(s_2) \Rightarrow c_1 + c_2 \in Cc(s_1 + s_2) \\
c_1 \in Cc(s_1) \wedge c_2 \in Cc(s_2) \Rightarrow c_1 - c_2 \in Cc(s_1 - s_2) \\
c_1 \in Cc(s_1) \wedge c_2 \in Cc(s_2) \Rightarrow c_1 \times c_2 \in Cc(s_1 \times s_2) \\
c \in Cc(s) \Rightarrow -c \in Cc(-s) \\
c \in Cc(s_1) \wedge c \in Cc(s_2) \Rightarrow c \in Cc(s_1 \sqcap s_2) \\
c \in Cc(s_1) \vee c \in Cc(s_2) \Rightarrow c \in Cc(s_1 \sqcup s_2)
\end{gathered}$$

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`, and satisfy the consistency conditions

$$\begin{array}{lcl}
\texttt{pos}(s) & \Leftrightarrow & \forall c \in Cc(s) : c > 0 \\
\texttt{neg}(s) & \Leftrightarrow & \forall c \in Cc(s) : c < 0 \\
\texttt{zer}(s) & \Leftrightarrow & Cc(s) = \{0\}
\end{array}$$

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. These functions satisfy the consistency conditions

$$\begin{array}{lcl}
\texttt{contains_pos}(s) & \Leftrightarrow & \exists c \in Cc(s) : c > 0 \\
\texttt{contains_neg}(s) & \Leftrightarrow & \exists c \in Cc(s) : c < 0 \\
\texttt{contains_zer}(s) & \Leftrightarrow & 0 \in Cc(s)
\end{array}$$

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 27 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 $3 = 2\texttt{P} + 3\texttt{R}$.

Example 28 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 $\{0 \leq 3\texttt{P} - \texttt{R}, 0 \leq 2\texttt{P} - 3\texttt{R}\}$.

Example 29 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 $\{3 = \texttt{P} + \texttt{R}\ ,\ 9 \leq 5\texttt{P}\} \leftrightarrow \{3 = \texttt{P} + \texttt{R}\ , 0 \leq 2\texttt{P} - 3\texttt{R}\}$.

```
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, 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'' ⊎ γ^0;
        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 7: Modified `Asplit_top` Algorithm for `LInt`

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 $\mathtt{pos}(\sigma[v])$, while an assigment $\sigma[v] := 0$ 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 7.

5.3 The Power Domain $2^{\mathtt{LInt}}$ 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 $2^{\mathtt{LInt}}$ in order to get the required accuracy. The definition of the various concepts (such as abstract multistore, concretization ordering and upper bound) in the domain $2^{\mathtt{LInt}}$ completely parallel the corresponding definitions in the domain $2^{\mathtt{LSign}}$. However, as $2^{\mathtt{LInt}}$ is an infinite domain, it becomes necessary to introduce a *widening* operator[6]. The use of widening operators was proposed in [11] and has been previously

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

used in domains such as type graphs for Prolog [55].

The design of a widening operator is both experimental and theoretical in nature. One the one hand, a widening operator should lead to efficient and accurate analyses; on the other hand, it should be possible to prove the termination and correctness of analyses using a widening operator.

Naive widenings for `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`. We begin by presenting the ideas behind the widening operator informally, and then present the operator in a more formal setting.

5.3.1 Informal Presentation

In our abstract interpretation framework for CLP, widening needs to be applied primarily in the following two situations:

1. when updating the output description of a predicate;

2. when updating the input description of a predicate.

This makes sure that there is no infinite sequence of refinements of any description. It should be clear from the above situations that the widening operator is applied to an old multistore α_{old} (e.g., the previous output of a predicate) and a new multistore α_{new} (e.g., the new output of a clause of the predicate) to give a new multistore α_{res} (e.g., the updated output of the predicate).

In the domain $2^{\texttt{LSign}}$, α_{res} is just the upper bound, i.e. set union of the abstract multistores α_{old} and α_{new}. Therefore, α_{res} is either exactly the same as α_{old} or a superset of α_{old}. In the former case, the sequence of refinements of the description reaches a stationary point. In the latter case, the number of `LSign` stores that are in the description increases. As the number of possible `LSign` stores for a given domain of variables is finite (the constant and each variable's coefficient can be assigned in four ways and there are exactly three multiplicities), the sequence must reach a stationary point.

The situation is not so simple in the domain $2^{\texttt{LInt}}$. As the number of possible `LInt` stores for a given domain of variables is infinite (due to infinitely many intervals), the sequence of descriptions can be refined infinitely often by the upper bound (adding more and more `LInt` stores to the multistore).

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. This is best motivated by an example.

Example 30 In order to keep the informal presentation simple, we shall consider the widening only for abstract stores - the formal presentation shows how to extend the definition to abstract multistores. Consider the mortgage example `mg/4`. After six iterations, the fixpoint algorithm to compute the output of `mg` has produced the following

abstract store as a possible output of `mg`.

$$\{\langle [0,0] < [1,1]\mathtt{T}, \mathtt{One}\rangle,\ \langle [0,0] \leq [1,1]\mathtt{P}, \mathtt{One}\rangle,\ \langle [3,5] = [1,1]\mathtt{T}, \mathtt{One}\rangle,$$
$$\langle [0,0] = [1.030301, 10510.100480]\mathtt{P} + [-51010.048, -3.0301]\mathtt{R} + [-10000, -1]\mathtt{B}, \mathtt{One}\rangle,$$
$$\langle [0,0] \leq [1,201]\mathtt{R} + [1,100]\mathtt{B}, \mathtt{One}\rangle, \langle [0,0] \leq [1,201]\mathtt{R} + [1,100]\mathtt{B}, \mathtt{Any}\rangle,$$
$$\langle [0,0] \leq [1.01, 102.01]\mathtt{P} + [-201, -1]\mathtt{R}, \mathtt{One}\rangle,$$
$$\langle [0,0] \leq [1.01, 102.01]\mathtt{P} + [-201, -1]\mathtt{R}, \mathtt{Any}\rangle\}$$

The seventh iteration produces the following abstract store as a possible output of `mg`.

$$\{\langle [0,0] < [1,1]\mathtt{T}, \mathtt{One}\rangle,\ \langle [0,0] \leq [1,1]\mathtt{P}, \mathtt{One}\rangle,\ \langle [3,6] = [1,1]\mathtt{T}, \mathtt{One}\rangle,$$
$$\langle [0,0] = [1.030301, 1061520.146432]\mathtt{P} + [-6152015.314944, -3.0301]\mathtt{R}$$
$$+[-999999.995904, -1]\mathtt{B}, \mathtt{One}\rangle,$$
$$\langle [0,0] \leq [1,201]\mathtt{R} + [1,100]\mathtt{B}, \mathtt{One}\rangle, \langle [0,0] \leq [1,201]\mathtt{R} + [1,100]\mathtt{B}, \mathtt{Any}\rangle,$$
$$\langle [0,0] \leq [1.01, 10303.01]\mathtt{P} + [-30301, -1]\mathtt{R}, \mathtt{One}\rangle,$$
$$\langle [0,0] \leq [1.01, 10303.01]\mathtt{P} + [-30301, -1]\mathtt{R}, \mathtt{Any}\rangle\}$$

We observe that there is a very close correspondence between the abstract constraints of the two stores. In particular, if the intervals in the constraints are replaced by their corresponding signs, then the two stores are identical. Hence the only difference in the two stores is that the coefficients of some constraints are different. As the two stores are the same (modulo replacement of intervals by signs), we need to guarantee that their intervals do not get refined infinitely often, while keeping the same signs. The basic idea of the widening is to take a pair of similar constraints from the two stores, see in what manner an interval coefficient in that constraint is changing, and accelerate that change. We illustrate this by considering the pair of similar constraints $\langle [3,5] = [1,1]\mathtt{T}, \mathtt{One}\rangle$ and $\langle [3,6] = [1,1]\mathtt{T}, \mathtt{One}\rangle$. The change in the constraint is that the right endpoint of the constant term is increasing. A constraint that approximates both these constraints and accelerates the direction in which the constraint is growing can be given by $\langle [3,+\infty) = [1,1]\mathtt{T}, \mathtt{One}\rangle$. This is the result of widening the pair of similar constraints. This widening can be performed on other pairs of similar constraints as well, giving the following widened abstract store after the seventh iteration of the fixpoint algorithm.

$$\{\langle [0,0] < [1,1]\mathtt{T}, \mathtt{One}\rangle,\ \langle [0,0] \leq [1,1]\mathtt{P}, \mathtt{One}\rangle,\ \langle [3,+\infty) = [1,1]\mathtt{T}, \mathtt{One}\rangle,$$
$$\langle [0,0] = [1.030301, +\infty)\mathtt{P} + (-\infty, -3.0301]\mathtt{R} + (-\infty, -1]\mathtt{B}, \mathtt{One}\rangle,$$
$$\langle [0,0] \leq [1,201]\mathtt{R} + [1,100]\mathtt{B}, \mathtt{One}\rangle, \langle [0,0] \leq [1,201]\mathtt{R} + [1,100]\mathtt{B}, \mathtt{Any}\rangle,$$
$$\langle [0,0] \leq [1.01, +\infty)\mathtt{P} + (-\infty, -1]\mathtt{R}, \mathtt{One}\rangle,$$
$$\langle [0,0] \leq [1.01, +\infty)\mathtt{P} + (-\infty, -1]\mathtt{R}, \mathtt{Any}\rangle\}$$

The widening for constraint stores is therefore applied only on stores that are the same (modulo replacement of intervals by signs) and it consists of widening the intervals of similar constraints in those stores.

To implement this intuition in a practical manner and to guarantee termination, our domain restricts attention to the sets of constraint stores to a subset of constraint stores satisfying two properties.

1. replacing intervals by the corresponding signs in an `LInt` store produces a valid `LSign` store (i.e. no duplicate constraints); and

2. replacing intervals by signs in a $2^{\texttt{LInt}}$ multistore produces a valid $2^{\texttt{LSign}}$ multistore (i.e. no duplicate stores).

Example 31 For instance, the constraint multistore

$$\{\langle [2,3] < [1,2]\texttt{P} + [1,3]\texttt{R}, \texttt{One}\rangle, \langle [1,4] < [2,3]\texttt{P} + [4,5]\texttt{R}, \texttt{One}\rangle\}$$

is not considered valid, because its two elements produce the same store

$$\langle \oplus < \oplus\texttt{P} + \oplus\texttt{R}, \texttt{One}\rangle$$

when replacing intervals by signs. On the other hand, both

$$\{\langle [2,3] < [1,2]\texttt{P} + [1,3]\texttt{R}, \texttt{One}\rangle, \langle [1,4] = [2,3]\texttt{P} + [4,5]\texttt{R}, \texttt{One}\rangle\}$$

and

$$\{\langle [2,3] < [1,2]\texttt{P} + [1,3]\texttt{R}, \texttt{One}\rangle, \langle [1,4] < [2,3]\texttt{P} + [-5,-4]\texttt{R}, \texttt{One}\rangle\}$$

are considered valid multistores.

It is important to notice that there are only finitely many `LSign` stores over some predefined set of variables. As a consequence, the above restrictions automatically impose an upper bound on the size of an $2^{\texttt{LInt}}$ multistore. Note however that the domain is still infinite because of the use of intervals.

When the abstract stores satisfy the above two conditions, it becomes reasonably simple to ensure termination of the analysis. The basic ideas are as follows.

1. The analyzer lets the $2^{\texttt{LInt}}$ abstract objects grow until the old and new versions are similar when viewed as `LSign` objects. This must happen because the analysis only accumulates more stores (through union) and there are finitely many $2^{\texttt{LInt}}$ stores on some given set of variables.

2. When the $2^{\texttt{LInt}}$ multistores stabilize when viewed as $2^{\texttt{LSign}}$ objects, the widening studies how the intervals are changing and accelerates these changings towards $-\infty, 0$ or ∞.[7] Since there are only finitely many bounds, this approach guarantees that, when the size of the object remains the same, the total number of $0, -\infty$, and ∞ symbols increases.

5.3.2 Formal Presentation

We now formally present the widening operator for the domain $2^{\texttt{LInt}}$. 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.

[7] More values could be selected (e.g. all the constants of the program) while preserving the termination, and this would improve the accuracy in some cases.

Definition 23 [Shape and Widening of Intervals] The shape of an interval s is defined as

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

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

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

The widening is defined only for intervals that have the same shape. 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 6 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 shape and widening for abstract constraints with multiplicity are natural extensions of the definition for intervals.

Definition 24 [Shape and Widening of Abstract Constraints With Multiplicity] The shape of an abstract constraint with multiplicity $\langle s_0 \ \delta \ \sum_{i=1}^{n} s_i x_i, \mu \rangle$ is defined as

$$\mathtt{shape}(\langle s_0 \ \delta \ \sum_{i=1}^{n} s_i x_i, \mu \rangle) = \langle \mathtt{shape}(s_0) \ \delta \ \sum_{i=1}^{n} \mathtt{shape}(s_i) x_i, \mu \rangle.$$

Let $\gamma_{old} = \langle s_0^{old} \ \delta \ \sum_{i=1}^{n} s_i^{old} x_i, \ \mu \rangle$ and $\gamma_{new} = \langle s_0^{new} \ \delta \ \sum_{i=1}^{n} s_i^{new} x_i, \ \mu \rangle$ be abstract constraints with multiplicity s.t. $\mathtt{shape}(\gamma_{new}) = \mathtt{shape}(\gamma_{old})$. Then

$$\gamma_{new} \nabla \gamma_{old} = \langle (s_0^{new} \nabla s_0^{old}) \ \delta \ \sum_{i=1}^{n} (s_i^{new} \nabla s_i^{old}) x_i, \ \mu \rangle$$

Once again, the widening is defined only for abstract objects that have the same shape. The following lemma is an easy consequence of the definition and the lemma for interval widening.

Lemma 7 Let γ_{old} and γ_{new} be abstract constraints with multiplicity s.t. $\texttt{shape}(\gamma_{new}) = \texttt{shape}(\gamma_{old})$. Then
(i) $\texttt{shape}(\gamma_{old}) = \texttt{shape}(\gamma_{new} \nabla \gamma_{old})$.
(ii) $\theta \in Cc(\gamma_{old}) \vee \theta \in Cc(\gamma_{new}) \Rightarrow \theta \in Cc(\gamma_{new} \nabla \gamma_{old})$.

Before defining the widening for abstract stores, it is necessary to define the notion of a normalized abstract store.

Definition 25 [Normalized Abstract Store] An abstract store β is said to be normalized if

$$\forall \gamma_1, \gamma_2 \in \beta : \texttt{shape}(\gamma_1) \neq \texttt{shape}(\gamma_2).$$

Given an abstract store β, it is possible to transform it into a normalized abstract store $\texttt{normal}(\beta)$ such that

$$\theta \in Cc(\beta) \Rightarrow \theta \in Cc(\texttt{normal}(\beta)).$$

Working with normalized abstract stores makes the correspondence between abstract stores and their shapes more easy to define.

Definition 26 [Shape and Widening of Abstract Stores] The shape of an normalized abstract store β is defined as

$$\texttt{shape}(\beta) = \{\texttt{shape}(\gamma) \mid \gamma \in \beta\}.$$

Let β_{old} and β_{new} be normalized abstract stores s.t. $\texttt{shape}(\beta_{new}) = \texttt{shape}(\beta_{old})$. Then

$$\beta_{new} \nabla \beta_{old} \;=\; \{\gamma_{new} \nabla \gamma_{old} \mid \gamma_{new} \in \beta_{new},\ \gamma_{old} \in \beta_{old},\ \texttt{shape}(\gamma_{new}) = \texttt{shape}(\gamma_{old})\}$$

Once again, the widening is only defined for abstract stores that have the same shape. The widening consists of applying widening over constraints to the pairs of constraints that have the same shape. The following lemma holds:

Lemma 8 Let β_{old} and β_{new} be normalized abstract stores s.t. $\texttt{shape}(\beta_{new}) = \texttt{shape}(\beta_{old})$. Then
(i) $\texttt{shape}(\beta_{old}) = \texttt{shape}(\beta_{new} \nabla \beta_{old})$ and $\beta_{new} \nabla \beta_{old}$ is normalized.
(ii) $\theta \in Cc(\beta_{old}) \vee \theta \in Cc(\beta_{new}) \Rightarrow \theta \in Cc(\beta_{new} \nabla \beta_{old})$.

Before defining the widening for abstract multistores, we need to define the notion of a normalized abstract multistore. In the following, we use the notation $\beta_1 \equiv \beta_2$ to mean that β_1 and β_2 are syntactically identical. This is different from the equality on abstract stores ($\beta_1 = \beta_2$) because the ordering on abstract stores is not a partial order, only a pre-order.

Definition 27 [Normalized Abstract Multistore] An abstract multistore α is said to be normalized if

- $\forall \beta \in \alpha : \beta$ is normalized

- $\forall \beta_1, \beta_2 \in \alpha : \texttt{shape}(\beta_1) \not\equiv \texttt{shape}(\beta_2)$

Given an abstract multistore α, it is possible to transform it into a corresponding normalized multistore $\texttt{normal}(\alpha)$ such that

$$\theta \in Cc(\alpha) \Rightarrow \theta \in Cc(\texttt{normal}(\alpha)).$$

Working with normalized abstract multistores simplifies the definition of the shape and widening.

Definition 28 [Shape and Widening of Abstract Multistores] The shape of an normalized abstract multistore α is defined as

$$\texttt{shape}(\alpha) = \{\texttt{shape}(\beta) \mid \beta \in \alpha\}.$$

Let α_{old} and α_{new} be normalized abstract multistores. Then

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

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 9 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 13 [Widening] Operation ∇ is a widening operator.

Sketch of Proof: The proof has two parts. The first part is to prove that the widened store does not decrease the concretization as compared to the concretizations of the old and new stores. This follows by Lemma 9 (ii):

$$\theta \in Cc(\alpha_{old}) \vee \theta \in Cc(\alpha_{new}) \Rightarrow \theta \in Cc(\alpha_{new} \nabla \alpha_{old}).$$

The second part of the proof is to show that given a sequence of normalized abstract multistores such that each element of the sequence is produced by widening the previous element with another store, the sequence is finite. This corresponds to not refining a description infinitely often. Let $\alpha'_0, \ldots, \alpha'_i, \ldots$ be a sequence of abstract multistores and $\alpha_0, \ldots, \alpha_i, \ldots$ a sequence of normalized abstract multistores defined as

$$\begin{array}{lcl} \alpha_0 & = & \texttt{normal}(\alpha'_0) \\ \alpha_{i+1} & = & \texttt{normal}(\alpha'_i) \nabla\ \alpha_i \quad (i \geq 0) \end{array}$$

The proof consists of showing that $\alpha_0, \ldots, \alpha_i, \ldots$ is stationary. The proof proceeds by defining a mapping from abstract multistores to a finite set of *measures* and proving that if the sequence is not at a stationary point, the measure of α_i increases monotonically with i. As there is only a finite number of measures, the sequence must reach a stationary point. □

5.4 Complete Example

We now present a complete example indicating how to compute the output and input descriptions in the $2^{\texttt{LInt}}$ domain, for the mortgage program `mg/4`. First we show the computation of the goal independent (or online) output.

Example 32 [Computation of `LInt` Output Description for `mg/4`] The abstract substitution describing the output of the first clause is

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

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

$$\begin{array}{l}\{\{\langle [0,0] < [1,1]\texttt{T}, \texttt{One}\rangle, \langle [0,0] \leq [1,1]\texttt{P}, \texttt{One}\rangle,\\ \quad \langle [0,0] = [1.01,1.01]\texttt{P} + [-1,-1]\texttt{R} + [-1,-1]\texttt{P1}, \texttt{One}\rangle, \langle [1,1] = [1,1]\texttt{T} + [-1,-1]\texttt{T1}, \texttt{One}\rangle\}\}\end{array}$$

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

$$\begin{array}{l}\{\{\langle [0,0] < [1,1]\texttt{T}, \texttt{One}\rangle, \langle [0,0] \leq [1,1]\texttt{P}, \texttt{One}\rangle,\\ \quad \langle [0,0] = [1.01,1.01]\texttt{P} + [-1,-1]\texttt{R} + [-1,-1]\texttt{P1}, \texttt{One}\rangle, \langle [1,1] = [1,1]\texttt{T} + [-1,-1]\texttt{T1}, \texttt{One}\rangle,\\ \quad \langle [0,0] = [1,1]\texttt{T1}, \texttt{One}\rangle, \langle [0,0] = [1,1]\texttt{P1} + [-1,-1]\texttt{B}, \texttt{One}\rangle\}\}\end{array}$$

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

$$\begin{array}{l}\{\{\langle [0,0] < [1,1]\texttt{T}, \texttt{One}\rangle, \langle [1,1] = [1,1]\texttt{T}, \texttt{One}\rangle,\\ \quad \langle [0,0] \leq [1,1]\texttt{P}, \texttt{One}\rangle, \langle [0,0] = [1.01,1.01]\texttt{P} + [-1,-1]\texttt{R} + [-1,-1]\texttt{B}, \texttt{One}\rangle\}\}\end{array}$$

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

$$\begin{array}{l}\{\\ \quad \{\langle [0,0] = [1,1]\texttt{T}, \texttt{One}\rangle, \langle [0,0] = [1,1]\texttt{P} + [-1,-1]\texttt{B}, \texttt{One}\rangle\},\\ \quad \{\langle [0,0] < [1,1]\texttt{T}, \texttt{One}\rangle, \langle [1,1] = [1,1]\texttt{T}, \texttt{One}\rangle,\\ \quad\quad \langle [0,0] \leq [1,1]\texttt{P}, \texttt{One}\rangle, \langle [0,0] = [1.01,1.01]\texttt{P} + [-1,-1]\texttt{R} + [-1,-1]\texttt{B}, \texttt{One}\rangle\}\\ \}\end{array}$$

In this case, the widening is just the union of mulstistores because there are no stores in the old output that have the same shape as a store in the new output. The process can

be iterated with the updated output of `mg`. After a few steps, the following multistore emerges as the output for `mg` (α_{old}):

$$
\begin{array}{l}
\{ \\
\quad \{\langle [0,0] = [1,1]\mathtt{T}, \mathtt{One}\rangle, \langle [0,0] = [1,1]\mathtt{P} + [-1,-1]\mathtt{B}, \mathtt{One}\rangle\}, \\
\quad \{\langle [0,0] < [1,1]\mathtt{T}, \mathtt{One}\rangle, \langle [1,1] = [1,1]\mathtt{T}, \mathtt{One}\rangle, \\
\quad\ \langle [0,0] \leq [1,1]\mathtt{P}, \mathtt{One}\rangle, \langle [0,0] = [1.01,1.01]\mathtt{P} + [-1,-1]\mathtt{R} + [-1,-1]\mathtt{B}, \mathtt{One}\rangle\} \\
\quad \{\langle [0,0] < [1,1]\mathtt{T}, \mathtt{One}\rangle, \langle [0,0] \leq [1,1]\mathtt{P}, \mathtt{One}\rangle, \langle [2,3] = [1,1]\mathtt{T}, \mathtt{One}\rangle, \\
\quad\ \langle [0,0] = [1.0201, 103.0301]\mathtt{P} + [-303.01, -2.01]\mathtt{R} + [-100,-1]\mathtt{B}, \mathtt{One}\rangle, \\
\quad\ \langle [0,0] \leq [1,201]\mathtt{R} + [1,100]\mathtt{B}, \mathtt{One}\rangle, \langle\ [0,0] \leq [1,201]\mathtt{R} + [1,100]\mathtt{B}, \mathtt{Any}\rangle\}, \\
\quad \{\langle [0,0] < [1,1]\mathtt{T}, \mathtt{One}\rangle, \langle [0,0] \leq [1,1]\mathtt{P}, \mathtt{One}\rangle, \langle [3,5] = [1,1]\mathtt{T}, \mathtt{One}\rangle, \\
\quad\ \langle [0,0] = [1.030301, 10510.100480]\mathtt{P} + [-51010.048, -3.0301]\mathtt{R} + [-10000,-1]\mathtt{B}, \mathtt{One}\rangle, \\
\quad\ \langle [0,0] \leq [1,201]\mathtt{R} + [1,100]\mathtt{B}, \mathtt{One}\rangle, \langle\ [0,0] \leq [1,201]\mathtt{R} + [1,100]\mathtt{B}, \mathtt{Any}\rangle, \\
\quad\ \langle [0,0] \leq [1.01, 102.01]\mathtt{P} + [-201,-1]\mathtt{R}, \mathtt{One}\rangle, \\
\quad\ \langle [0,0] \leq [1.01, 102.01]\mathtt{P} + [-201,-1]\mathtt{R}, \mathtt{Any}\rangle\} \\
\}
\end{array}
$$

Using this in the second clause produces the following multistore as a new output for `mg` (α_{new}):

$$
\begin{array}{l}
\{ \\
\quad \{\langle [0,0] = [1,1]\mathtt{T}, \mathtt{One}\rangle, \langle [0,0] = [1,1]\mathtt{P} + [-1,-1]\mathtt{B}, \mathtt{One}\rangle\}, \\
\quad \{\langle [0,0] < [1,1]\mathtt{T}, \mathtt{One}\rangle, \langle [1,1] = [1,1]\mathtt{T}, \mathtt{One}\rangle, \\
\quad\ \langle [0,0] \leq [1,1]\mathtt{P}, \mathtt{One}\rangle, \langle [0,0] = [1.01,1.01]\mathtt{P} + [-1,-1]\mathtt{R} + [-1,-1]\mathtt{B}, \mathtt{One}\rangle\} \\
\quad \{\langle [0,0] < [1,1]\mathtt{T}, \mathtt{One}\rangle, \langle [0,0] \leq [1,1]\mathtt{P}, \mathtt{One}\rangle, \langle [2,3] = [1,1]\mathtt{T}, \mathtt{One}\rangle, \\
\quad\ \langle [0,0] = [1.0201, 103.0301]\mathtt{P} + [-303.01, -2.01]\mathtt{R} + [-100,-1]\mathtt{B}, \mathtt{One}\rangle, \\
\quad\ \langle [0,0] \leq [1,201]\mathtt{R} + [1,100]\mathtt{B}, \mathtt{One}\rangle, \langle\ [0,0] \leq [1,201]\mathtt{R} + [1,100]\mathtt{B}, \mathtt{Any}\rangle\}, \\
\quad \{\langle [0,0] < [1,1]\mathtt{T}, \mathtt{One}\rangle, \langle [0,0] \leq [1,1]\mathtt{P}, \mathtt{One}\rangle, \langle [3,6] = [1,1]\mathtt{T}, \mathtt{One}\rangle, \\
\quad\ \langle [0,0] = [1.030301, 1061520.146432]\mathtt{P} + [-6152015.314944, -3.0301]\mathtt{R} \\
\qquad\qquad +[-999999.995904, -1]\mathtt{B}, \mathtt{One}\rangle, \\
\quad\ \langle [0,0] \leq [1,201]\mathtt{R} + [1,100]\mathtt{B}, \mathtt{One}\rangle, \langle\ [0,0] \leq [1,201]\mathtt{R} + [1,100]\mathtt{B}, \mathtt{Any}\rangle, \\
\quad\ \langle [0,0] \leq [1.01, 10303.01]\mathtt{P} + [-30301,-1]\mathtt{R}, \mathtt{One}\rangle, \\
\quad\ \langle [0,0] \leq [1.01, 10303.01]\mathtt{P} + [-30301,-1]\mathtt{R}, \mathtt{Any}\rangle\} \\
\}
\end{array}
$$

At this point the widening needs to be applied and produces the following multistore as

the updated output of mg ($\alpha_{new} \nabla \alpha_{old}$):

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

Iterating the output computation once more produces the same abstract substitution indicating that this abstract multistore is the fixpoint of the computation and represents the output of the predicate mg.

We now illustrate the computation of the input description, when the top level query is such that both P and R are fixed.

Example 33 [Computation of LInt Input Description for mg/4] The abstract substitution describing a top level query such that P and R are fixed, is given as

$$\{\{\langle (-\infty, +\infty) = [1,1]\mathtt{P}, \mathtt{One}\rangle, \langle (-\infty, +\infty) = [1,1]\mathtt{R}, \mathtt{One}\rangle\}\}$$

When the second clause is executed with this input, the abstract substitution for the program point just before the recursive call to mg is

$$
\begin{array}{l}
\{\{\langle (-\infty, +\infty) = [1,1]\mathtt{P}, \mathtt{One}\rangle, \langle (-\infty, +\infty) = [1,1]\mathtt{R}, \mathtt{One}\rangle, \\
\quad \langle [0,0] < [1,1]\mathtt{T}, \mathtt{One}\rangle, \langle [0,0] \leq [1,1]\mathtt{P}, \mathtt{One}\rangle, \\
\quad \langle [0,0] = [1.01,1.01]\mathtt{P} + [-1,-1]\mathtt{R} + [-1,-1]\mathtt{P1}, \mathtt{One}\rangle, \langle [1,1] = [1,1]\mathtt{T} + [-1,-1]\mathtt{T1}, \mathtt{One}\rangle\}\}
\end{array}
$$

When restricted to the variables of the literal mg(P1,T1,R,B), this gives the abstract substitution

$$\{\{\langle [-1,-1] < [1,1]\mathtt{T1}, \mathtt{One}\rangle, \langle (-\infty, +\infty) = [1,1]\mathtt{R}, \mathtt{One}\rangle, \langle (-\infty, +\infty) = [1,1]\mathtt{P1} + [1,1]\mathtt{R}, \mathtt{One}\rangle\}\}$$

which on normalization gives the following input substitution for the recursive call to mg

$$\{\{\langle [-1,-1] < [1,1]\mathtt{T}, \mathtt{One}\rangle, \langle (-\infty, +\infty) = [1,1]\mathtt{R}, \mathtt{One}\rangle, \langle (-\infty, +\infty) = [1,1]\mathtt{P} + [1,1]\mathtt{R}, \mathtt{One}\rangle\}\}$$

The widening of this with the previous input gives the following abstract substitution as the updated input to mg

$$
\begin{array}{l}
\{ \\
\quad \{\langle (-\infty, +\infty) = [1,1]\mathtt{P}, \mathtt{One}\rangle, \langle (-\infty, +\infty) = [1,1]\mathtt{R}, \mathtt{One}\rangle\}, \\
\quad \{\langle [-1,-1] < [1,1]\mathtt{T}, \mathtt{One}\rangle, \langle (-\infty, +\infty) = [1,1]\mathtt{R}, \mathtt{One}\rangle, \langle (-\infty, +\infty) = [1,1]\mathtt{P} + [1,1]\mathtt{R}, \mathtt{One}\rangle\} \\
\}
\end{array}
$$

In this case, the widening of the multistores just corresponds to their union because none of the stores have the same shape. The second clause can now be reexecuted with the updated input to continue the process of computing the input description. After some stages of the computation, the following emerges as the input description to `mg` (α_{old}):

$$
\begin{array}{l}
\{ \\
\quad \{\langle (-\infty,+\infty) = [1,1]\mathtt{P}, \mathtt{One}\rangle, \langle (-\infty,+\infty) = [1,1]\mathtt{R}, \mathtt{One}\rangle\}, \\
\quad \{\langle [-1,-1] < [1,1]\mathtt{T}, \mathtt{One}\rangle, \langle (-\infty,+\infty) = [1,1]\mathtt{R}, \mathtt{One}\rangle, \langle (-\infty,+\infty) = [1,1]\mathtt{P} + [1,1]\mathtt{R}, \mathtt{One}\rangle\} \\
\quad \{\langle [-3,-1] < [1,1]\mathtt{T}, \mathtt{One}\rangle, \langle [-3,-1] < [1,1]\mathtt{T}, \mathtt{Any}\rangle, \langle (-\infty,+\infty) = [1,1]\mathtt{R}, \mathtt{One}\rangle, \\
\qquad \langle (-\infty,+\infty) \leq [-201,-1]\mathtt{R}, \mathtt{One}\rangle, \langle (-\infty,+\infty) \leq [-201,-1]\mathtt{R}, \mathtt{Any}\rangle, \\
\qquad \langle (-\infty,+\infty) = [1,100]\mathtt{P} + [2.01,303.01]\mathtt{R}, \mathtt{One}\rangle\} \\
\quad \{\langle [-5,-1] < [1,1]\mathtt{T}, \mathtt{One}\rangle, \langle [-5,-1] < [1,1]\mathtt{T}, \mathtt{Any}\rangle, \langle (-\infty,+\infty) = [1,1]\mathtt{R}, \mathtt{One}\rangle, \\
\qquad \langle (-\infty,+\infty) \leq [-201,-1]\mathtt{R}, \mathtt{One}\rangle, \langle (-\infty,+\infty) \leq [-201,-1]\mathtt{R}, \mathtt{Any}\rangle, \\
\qquad \langle [0,0] \leq [1,100]\mathtt{P} + [1,201]\mathtt{R}, \mathtt{One}\rangle, \langle [0,0] \leq [1,100]\mathtt{P} + [1,201]\mathtt{R}, \mathtt{Any}\rangle, \\
\qquad \langle (-\infty,+\infty) = [1,10000]\mathtt{P} + [3.0301,51010.048]\mathtt{R}, \mathtt{One}\rangle\} \\
\}
\end{array}
$$

Using this as the input to the second clause and recomputing the input for the recursive clause gives the following new input (α_{new}):

$$
\begin{array}{l}
\{ \\
\quad \{\langle [-1,-1] < [1,1]\mathtt{T}, \mathtt{One}\rangle, \langle (-\infty,+\infty) = [1,1]\mathtt{R}, \mathtt{One}\rangle, \langle (-\infty,+\infty) = [1,1]\mathtt{P} + [1,1]\mathtt{R}, \mathtt{One}\rangle\} \\
\quad \{\langle [-2,-1] < [1,1]\mathtt{T}, \mathtt{One}\rangle, \langle [-2,-1] < [1,1]\mathtt{T}, \mathtt{Any}\rangle, \langle (-\infty,+\infty) = [1,1]\mathtt{R}, \mathtt{One}\rangle, \\
\qquad \langle (-\infty,+\infty) \leq [-1,-1]\mathtt{R}, \mathtt{One}\rangle, \langle (-\infty,+\infty) \leq [-201,-1]\mathtt{R}, \mathtt{Any}\rangle, \\
\qquad \langle (-\infty,+\infty) = [1,1]\mathtt{P} + [2.01,2.01]\mathtt{R}, \mathtt{One}\rangle\} \\
\quad \{\langle [-6,-1] < [1,1]\mathtt{T}, \mathtt{One}\rangle, \langle [-6,-1] < [1,1]\mathtt{T}, \mathtt{Any}\rangle, \langle (-\infty,+\infty) = [1,1]\mathtt{R}, \mathtt{One}\rangle, \\
\qquad \langle (-\infty,+\infty) \leq [-201,-1]\mathtt{R}, \mathtt{One}\rangle, \langle (-\infty,+\infty) \leq [-201,-1]\mathtt{R}, \mathtt{Any}\rangle, \\
\qquad \langle [0,0] \leq [1,10000]\mathtt{P} + [1,30301]\mathtt{R}, \mathtt{One}\rangle, \langle [0,0] \leq [1,10000]\mathtt{P} + [1,30301]\mathtt{R}, \mathtt{Any}\rangle, \\
\qquad \langle (-\infty,+\infty) = [1,999999.995904]\mathtt{P} + [3.0301,6152014.790656]\mathtt{R}, \mathtt{One}\rangle\} \\
\}
\end{array}
$$

The updated input ($\alpha_{new} \nabla \alpha_{old}$) becomes

$$
\begin{array}{l}
\{ \\
\quad \{\langle (-\infty,+\infty) = [1,1]\mathtt{P}, \mathtt{One}\rangle, \langle (-\infty,+\infty) = [1,1]\mathtt{R}, \mathtt{One}\rangle\}, \\
\quad \{\langle [-1,-1] < [1,1]\mathtt{T}, \mathtt{One}\rangle, \langle (-\infty,+\infty) = [1,1]\mathtt{R}, \mathtt{One}\rangle, \langle (-\infty,+\infty) = [1,1]\mathtt{P} + [1,1]\mathtt{R}, \mathtt{One}\rangle\} \\
\quad \{\langle [-3,-1] < [1,1]\mathtt{T}, \mathtt{One}\rangle, \langle [-3,-1] < [1,1]\mathtt{T}, \mathtt{Any}\rangle, \langle (-\infty,+\infty) = [1,1]\mathtt{R}, \mathtt{One}\rangle, \\
\qquad \langle (-\infty,+\infty) \leq [-201,-1]\mathtt{R}, \mathtt{One}\rangle, \langle (-\infty,+\infty) \leq [-201,-1]\mathtt{R}, \mathtt{Any}\rangle, \\
\qquad \langle (-\infty,+\infty) = [1,100]\mathtt{P} + [2.01,303.01]\mathtt{R}, \mathtt{One}\rangle\} \\
\quad \{\langle (-\infty,-1] < [1,1]\mathtt{T}, \mathtt{One}\rangle, \langle (-\infty,-1] < [1,1]\mathtt{T}, \mathtt{Any}\rangle, \langle (-\infty,+\infty) = [1,1]\mathtt{R}, \mathtt{One}\rangle, \\
\qquad \langle (-\infty,+\infty) \leq [-201,-1]\mathtt{R}, \mathtt{One}\rangle, \langle (-\infty,+\infty) \leq [-201,-1]\mathtt{R}, \mathtt{Any}\rangle, \\
\qquad \langle [0,0] \leq [1,+\infty)\mathtt{P} + [1,+\infty)\mathtt{R}, \mathtt{One}\rangle, \langle [0,0] \leq [1,+\infty)\mathtt{P} + [1,+\infty)\mathtt{R}, \mathtt{Any}\rangle, \\
\qquad \langle (-\infty,+\infty) = [1,+\infty)\mathtt{P} + [3.0301,+\infty)\mathtt{R}, \mathtt{One}\rangle\} \\
\}
\end{array}
$$

This is the fixpoint of the computation and represents all the possible inputs to `mg` when the top level query is such that `P` and `R` are fixed.

5.5 Discussion

We now illustrate how 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 $\beta = \{\langle \oplus = \oplus x, One\rangle, \langle \oplus < \oplus x, One\rangle\}$. While θ is satisfiable, `Ais_sat`(β) is false because the abstract store obtained by projecting all the variables is $\{\langle \top < 0, One\rangle\}$. Moving to the domain `LInt`, the abstraction of θ is $\beta' = \{\langle [3,3] = [1,1]x, One\rangle, \langle [2,2] < [1,1]x, One\rangle\}$. Projecting all the variables gives the abstract store $\{\langle [-1,-1] < 0, One\rangle\}$ and so `Ais_sat`(β') 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 preliminary evidence that the domains are sufficiently precise to perform the intended optimizations. It shows that an optimizing compiler can indeed use the domains to 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 `LInt` instead of `LSign`.

It should be stressed that the experimental results presented in this section are included only as some preliminary evidence that the domains are sufficiently precise and can be implemented, the main focus of the paper being on the theoretical treatment. As discussed previously in the paper, it is far from obvious to reorder even simple programs because conditional satisfiability is a non-trivial operation: it is thus important to discuss this issue. Also, 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 [21] should be helpful here to obtain a fast implementation. The design of a fast incremental compiler is, of course, outside the scope of this paper.

The section is organized as follows. Section 6.1 presents the benchmarks used in the experimental results. The execution time for optimized and unoptimized versions of the benchmarks is compared in Section 6.2. The compilation times using the `LSign` and `LInt` analyses is presented in Section 6.3. Note that the compiler also uses the domain `Prop`.

6.1 The Benchmarks

Our benchmarks are relatively small, since most large `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)` can be used either to check if its argument is a non-negative integer or to generate the non-negative integers. In the results, it is used to verify if 25000 is an integer and to generate the first 250 integers. The next three programs, `Exp(N,E)`, `Sum(N,S)` and `Fibonacci(N,F)` are programs that compute 2^{ttN}, the sum of the first `N` integers, and the `N`th Fibonacci number, respectively. Once again, these

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 inital and final y values
Triangular, 2000		Solve N equations, N = 2000
4000		Solve N equations, N = 4000
8000		Solve N equations, N = 8000

Table 1: Test Programs: Description of Usage in Various Modes.

benchmarks are used in various modes such as computing the sum of the first `N` integers, determining `N` such that a given number `F` is the $\texttt{N}^{th}$ Fibonacci number, generating all pairs of numbers $\langle \texttt{N}, 2^{\texttt{N}} \rangle$ for $0 \leq \texttt{N} \leq 25$ etc. `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. In our benchmarks, 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` [40] is a program solving the ordinary differential equation $y' = t$ numerically using Euler method. `Triangular` is a benchmark that involves simultaneously solving a sparse system of `N` equations, subsystems of which are in upper triangular form. Table 1 summarizes the usage modes of the various test programs. Some of the programs (`Mortgage` and `Ode-Euler`) are syntactically non-linear, but the queries make them linear at runtime.

6.2 Execution Time

The benchmarks are used to compare the optimizing compiler with the standard compiler. 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 [53] and is close to the architecture of the CLP($\Re$) system [26].

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 2: Comparison of Running Times in Milliseconds: Optimized vs. Unoptimized.

The runtime system uses infinite precision integers to guarantee numerical stability. Table 2 reports the computation times (in milliseconds on a SUN-SPARC-10) for the optimized and unoptimized versions of the benchmarks for various modes. It also gives the ratio of the unoptimized execution time to the optimized execution time. This number is called the *speedup* and it quantifies the benefit of optimization, i.e. a larger speedup implies a greater benefit due to optimization. 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 √ indicates that the the unoptimized program utilized the constraint solver to perform constraint solving, while the optimized program bypassed the constraint solver entirely.

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.

Program	Mode	Unopt.	Reordering (1)	Removal (2)	Refinement (3)	Total (1) + (2) + (3)
Integer	f	100	120	20	10	150
	u	100	400	10	10	420
Exp	fu	150	510	10	10	530
	uf	120	690	10	10	710
	uu	110	980	10	20	1010
Sum	fu	160	1200	10	10	1220
	uf	130	1470	20	20	1510
	uu	180	1530	10	10	1550
Fibonacci	fu	90	4280	30	20	4330
	uf	110	7900	90	80	8070
	uu	100	6150	70	50	6270
Mortgage	fffu	120	1420	20	20	1460
(Linear)	fufu	100	3060	30	10	3100
Mortgage	ffffu	160	3350	50	40	3440
(Nonlinear)	fuffu	240	9480	50	30	9560
Ode-Euler	ffffu	110	1680	40	10	1730
	fufff	110	1650	30	20	1700
	fuffu	140	1140	20	20	1180
Triangular, 2000		100	360	20	10	390
4000		120	480	30	10	520
8000		160	2320	80	50	2450

Table 3: Optimization Times in Milliseconds: `LSign`.

6.3 Compilation Time

We now investigate the price the pay to move from `LSign` to `LInt`. Table 3 gives the optimization time (in milliseconds) for the benchmarks along with the compilation time if no optimizations are performed (column Unopt.). The numbers are given for a version of the compiler in which the `LSign` and `Prop` analyses are used. The optimization time is given separately for each of the three phases, as well as for the total of the three phases. Note that, at this point, no effort has been spent in making the analyses or the optimization algorithms fast. In particular, reordering is particular expensive due to the non-incremental nature of the compiler.

In order to experimentally quantify the penalty paid for using intervals instead of signs, it is also useful to compare the optimization times for the benchmarks when `LInt` is used in the analyzer instead of `LSign`. Table 4 gives the total optimization times (in milliseconds) for the benchmarks when the analysis is performed using `LInt` and also when the analysis is performed using `LSign`. The ratio of the optimization times for `LInt` vs. `LSign` is also given for comparison. For our benchmarks, the average penalty paid by the `LInt` analysis over the `LSign` analysis is a factor of 7.39.

7 Conclusion

This paper has studied two abstract domains, `LSign` and `LInt`, which can be used at compile time when it is safe to reorder CLP programs. The domain `LSign` was originally proposed in [38] and further studied in [46]. Its fundamental ideas consist of abstracting

Program	Mode	LInt (1)	LSign (2)	Ratio (1)/(2)
Integer	f	800	150	5.33
	u	2150	420	5.12
Exp	fu	4110	530	7.75
	uf	5140	710	7.24
	uu	7020	1010	6.95
Sum	fu	8060	1220	6.61
	uf	10340	1510	6.88
	uu	11180	1550	7.21
Fibonacci	fu	43910	4330	10.14
	uf	120750	8070	14.96
	uu	76610	6270	12.22
Mortgage	fffu	14410	1460	9.87
(Linear)	fufu	42420	3100	13.68
Mortgage	ffffu	25070	3440	7.29
(Nonlinear)	fuffu	62180	9560	6.50
Ode-Euler	ffffu	11110	1730	6.42
	fufff	11070	1700	6.51
	fuffu	7200	1180	6.10
Triangular, 2000		1350	390	3.46
4000		1510	520	2.90
8000		3920	2450	1.60
Ar. Mean				7.39

Table 4: Comparison of Optimization Times: `LInt` vs. `LSign`.

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. As far as `LSign` is concerned, the paper redefined the domain of [38] to simplify the correctness proofs. It proposes a new ordering capturing the intended meaning of [38] and shows that two `LSign` descriptions can be ordered in polynomial time. It proposed a more precise algorithm for projection whose essential structure is modeled after [38]. Finally, it discussed 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, was shown to raise some subtle practical issues which are studied at length. As far as `LInt` is concerned, the paper presented the first presentation of the domain, and proposed a precise widening operator. The operator has the original property of using `LSign` to guide the widening process. 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 [21]. Equally important is the enhancement of the domains to support first-order terms. The developments will make it possible to apply the domains on large programs and to give a definitive assessment on their practical value.

References

[1] A.V. Aho, R. Sethi and J.D. Ullman. *Compilers Principles Techniques and Tools*, Addison Wesley, 1986.

[2] R. Barbuti, R. Giacobazzi and G. Levi. A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(1):133–181, January 1993.

[3] 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.

[4] 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.

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

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

[7] C. Codognet, P. Codognet and J.M. Corsini. Abstract Interpretation of Concurrent Logic Languages. in *Proc. North American Conference on Logic Programming (NACLP-90)*, Austin, TX, October 1990.

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

[9] A. Cortesi, G. Fil, and W. H. Winsborough, Prop revisited: Propositional Formula as Abstract Domain for Groundness Analysis. In *Proc. Sixth Annual IEEE Symposium on Logic in Computer Science*, 322–327, Amsterdam, The Netherlands, IEEE Computer Society Press, 1991.

[10] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of Abstract Domains for Logic Programming. In *Proc. 21st Annual ACM SIGPLAN–SIGACT Symposium on Principles Of Programming Languages*, 227–239, Portland, OR, January 1994.

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

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

[19] F. Giannotti and M. Hermenegildo. A Technique for Recursive Invariance Detection and Selective Program Specialization. In *Proc. 3rd. Int'l Symposium on Programming Language Implementation and Logic Programming*, number 528 in LNCS, pages 323–335. Springer-Verlag, August 1991.

[20] 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.

[21] M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey Incremental Analysis of Logic Programs. In *1995 International Conference on Logic Programming*, Japan, June, 1995.

[22] N. Heintze and J. Jaffar. An Engine for Logic Program Analysis. In *IEEE 7th Annual Symposium on Logic in Computer Science*, pages 318–328, 1992.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] N. Joldrgensen, 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.

[28] T. Kanamori and T. Kawamura. Analysing Success Patterns of Logic Programs by Abstract Hybrid Interpretation. Technical Report, ICOT, 1987.

[29] A.D. Kelly, A.D. Macdonald, K. Marriott, H. Sndergaard, P.J. Stuckey, and R.H C. Yap. An Optimizing Compiler for CLP(R). In U. Montanari, F. Rossi (Eds.) *Proc. First Int. Conference on Principles and Practice of Constraint Programming - CP'95*, Lecture Notes in Computer Science, Vol. 976, 222–239, Springer 1995

[30] A.D. Kelly, A.D. Macdonald, K. Marriott, P.J. Stuckey, and R.H C. Yap. Effectiveness of Optimizing Compilation for CLP(R) In Michael J. Maher (Ed.) *Proc. Joint International Conference and Syposium on Logic Programming*, 37–51, MIT Press, 1996.

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

[32] 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.

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

[34] K. Marriott and H. Sondergaard. Semantics-based Dataflow Analysis of Logic Programs. In *Information Processing-89*, 601–606, San Francisco, CA, 1989.

[35] K. Marriott, H. Sondergaard, and N.D. Jones. Denotational Abstract Interpretation of Logic Programs, *ACM Transactions on Programming Languages and Systems*, 16(3): 607-648, 1994.

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

[37] 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.

[38] 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.

[39] C. Mellish. *Abstract Interpretation of Prolog Programs* In Ehud Y. Shapiro (Ed.) *Proc. Third Int.l Conference on Logic Programming.* Lecture Notes in Computer Science, Vol. 225, 463–474, Springer, 1986.

[40] 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.

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

[42] R.A. O'Keefe. Finite Fixed-Point Problems. in Fourth Intl. Conf. on Logic Programming, 729–743, Melbourne, Australia, 1987.

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

[44] 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.

[45] 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.

[46] V. Ramachandran and P. Van Hentenryck. LSign Reordered. In Alan Mycroft (Ed.) *Proc. Static Analysis Symposium (SAS-95).* Lecture Notes in Computer Science, Vol. 983, 330–347, Springer Verlag, 1995.

[47] P. Refalo, and P.Van Hentenryck. CLP(Rlin) Revised *Proc. Joint International Conference and Syposium on Logic Programming*, 22–36, MIT Press, 1996.

[48] V. Ramachandran. An Optimizing Compiler for `CLP(`$\Re_{Lin}$`)`, PhD thesis, Brown University, forthcoming.

[49] A. Taylor. LIPS on a MIPS:Results from a Prolog Compiler for a RISC. In *Logic Programming: Proc. Seventh Intl. Conf.*, 174–185, MIT Press, 1990.

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

[51] 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.

[52] P. Van Hentenryck and V. Ramachandran. Backtracking without Trailing in CLP($\Re_{Lin}$). In *Proc. ACM-SIGPLAN Conf. on Programming Language Design and Implementation (PLDI-94)*, Orlando, FL, June 1994.

[53] P. Van Hentenryck and V. Ramachandran. Backtracking without Trailing in CLP($\Re_{Lin}$). *ACM Transactions on Programming Languages and Systems*, July 1995.

[54] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type Analysis of Prolog Using Type Graphs. In *Proc. ACM-SIGPLAN Conf. on Programming Language Design and Implementation (PLDI-94)*, Orlando, FL, June 1994.

[55] 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.

[56] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of `Prop`. *Journal of Logic Programming*, 23(3): 237–278, 1995.

[57] P. Van Roy and A. Despain. The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler. In *Logic Programming: Proc. North American Conf. 1990*, 501–515, MIT Press, 1990.

[58] W. Winsborough. Multiple Specialization using Minimal Function Graph Semantics. *Journal of Logic Programming*, 13(2-3):259–290, 1992.

Appendix: Proofs of the Results

Ordering

Lemma 10 [Composition of Ordering Functions] Let $\beta_1, \beta_2, \beta_3$ be abstract stores, let f be an ordering function of β_1 to β_2, and let g be an ordering function of β_2 to β_3. The composition $g \circ f$ is an ordering function of β_1 to β_3.

Proof: The three properties of ordering functions hold.

1. $\forall \gamma \in \beta_1 : \gamma \sqsubseteq f(\gamma) \sqsubseteq g(f(\gamma))$.
2. By hypothesis, we have $\forall \gamma_1, \gamma_2 \in \beta_1 : \gamma_1 \neq \gamma_2 \Rightarrow f(\gamma_1) \neq f(\gamma_2) \vee f(\gamma_1) \in \texttt{Indef}(\beta_2)$ and $\forall \gamma_1, \gamma_2 \in \beta_2 : \gamma_1 \neq \gamma_2 \Rightarrow g(\gamma_1) \neq g(\gamma_2) \vee g(\gamma_1) \in \texttt{Indef}(\beta_3)$. By combination of the two statements and $f(\gamma_1) \sqsubseteq (g \circ f)(\gamma_1)$, we obtain $\forall \gamma_1, \gamma_2 \in \beta_1 : \gamma_1 \neq \gamma_2 \Rightarrow (g \circ f)(\gamma_1) \neq (g \circ f)(\gamma_2) \vee (g \circ f)(\gamma_1) \in \texttt{Indef}(\beta_3)$.
3. By hypothesis, we have $\texttt{Def}(\beta_3) \subseteq \mathit{range}(g)$. By property 1 of Definition 9, there exist $\gamma_1, \ldots, \gamma_n$ in $\texttt{Def}(\beta_2)$ such that $\{f(\gamma_1), \ldots, f(\gamma_n)\} = \texttt{Def}(\beta_3)$. The result follows from the hypothesis that $\texttt{Def}(\beta_2) \subseteq \mathit{range}(f)$.

□

Theorem 1 [Ordering Relation] $\sqsubseteq$: $AS_D \times AS_D$ is a pre-order.

Proof: Reflexivity follows by choosing the identity function as ordering function. Transitivity follows by Lemma 10. □

Lemma 11 Let γ be an abstract constraint with multiplicity and θ a concrete constraint store. $\theta \in \mathit{Cc}(\gamma) \Rightarrow \forall \lambda \in \theta : \lambda \in \mathit{Cc}(\mathit{cons}(\gamma))$.

Proof: Immediate consequence of Definition 6. □

Lemma 3 [Lifting Function Lemma] Let β be an abstract store and θ a concrete constraint store. $\theta \in \mathit{Cc}_i(\beta)$ if and only if there exists a lifting function f of θ to β.

Proof: By induction on $|\beta|$.

Basis: $\beta = \emptyset$. If $\theta \in \mathit{Cc}_i(\beta)$ then $\theta = \emptyset$ by Definition 7 and we choose the empty function as a lifting function. Conversely, if there exists a function $f : \theta \to \beta$, then $\theta = \emptyset$ since $\beta = \emptyset$ and f must be the empty function. The result follows from Definition 7.
Induction Step: Assume that the hypothesis holds for all abstract stores whose cardinality is not greater than n. We show that it holds for abstract stores of cardinality $n+1$.

Consider an abstract store β satisfying $|\beta| = n + 1$ and let β be $\{\gamma\} \cup \beta'$, where $\gamma \notin \beta'$ and $|\beta'| = n$. By Definition 7,

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

($\Rightarrow$) Let $\theta \in Cc_i(\beta)$. Hence, $\theta = \theta_1 \underline{\cup} \theta'$, 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

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

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

1. $\forall \lambda \in \theta' : \lambda \in Cc(cons(f'(\lambda)))$ by Definition 11
   $\forall \lambda \in \theta_1 : \lambda \in Cc(cons(\gamma))$ by Lemma 11
   $\Rightarrow \forall \lambda \in \theta : \lambda \in Cc(cons(f(\lambda)))$.

2. $\forall \lambda_1, \lambda_2 \in \theta' : \lambda_1 \neq \lambda_2 \Rightarrow f'(\lambda_1) \neq f'(\lambda_2) \vee f'(\lambda_1) \in \texttt{Indef}(\beta')$ by Definition 11
   $\Rightarrow \forall \lambda_1, \lambda_2 \in \theta' : \lambda_1 \neq \lambda_2 \Rightarrow f(\lambda_1) \neq f(\lambda_2) \vee f(\lambda_1) \in \texttt{Indef}(\beta)$ since $\texttt{Indef}(\beta') \subseteq \texttt{Indef}(\beta)$

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

   $\forall \lambda_1, \lambda_2 \in \theta_1 : \lambda_1 \neq \lambda_2 \Rightarrow \gamma$ is indefinite by Definition 6
   $\Rightarrow \forall \lambda_1, \lambda_2 \in \theta_1 : \lambda_1 \neq \lambda_2 \Rightarrow f(\lambda_1) \in \texttt{Indef}(\beta)$ since $\gamma = f(\lambda_1)$
3. $\texttt{Def}(\beta') \subseteq range(f')$ by Definition 11
   γ is not definite $\Rightarrow \texttt{Def}(\beta) = \texttt{Def}(\beta') \wedge range(f') \subseteq range(f)$
   $\Rightarrow \texttt{Def}(\beta) \subseteq range(f)$
   γ is definite $\Rightarrow \texttt{Def}(\beta) = \{\gamma\} \cup \texttt{Def}(\beta') \wedge \{\gamma\} \cup range(f') = range(f)$
   $\Rightarrow \texttt{Def}(\beta) \subseteq range(f)$.

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

Case γ is not definite. Let $\lambda_1, \ldots, \lambda_m$ be all elements in θ such that $f(\lambda_i) = \gamma$ $(1 \leq i \leq m)$. $\lambda_i \in Cc(cons(\gamma))$ $(1 \leq i \leq m)$ by Definition 11 and $\{\lambda_1, \ldots, \lambda_m\} \in Cc(\gamma)$ by Definition 6. Let $\theta = \{\lambda_1, \ldots, \lambda_m\} \underline{\cup} \theta'$ and consider the function $f' : \theta' \to \beta'$ defined by $f'(\lambda) = f(\lambda)$ for all $\lambda \in \theta'$. We show that f' is a lifting function of θ' to β'.

1. $\forall \lambda \in \theta'$: $\lambda \in Cc(cons(f'(\lambda)))$ by Definition 11.

2. $\forall \lambda_1, \lambda_2 \in \theta' : \lambda_1 \neq \lambda_2 \Rightarrow f(\lambda_1) \neq f(\lambda_2) \vee f(\lambda_1) \in \texttt{Indef}(\beta)$ by Definition 11
   $\forall \lambda_1, \lambda_2 \in \theta' : \lambda_1 \neq \lambda_2 \Rightarrow f'(\lambda_1) \neq f'(\lambda_2) \vee f'(\lambda_1) \in \texttt{Indef}(\beta')$
   since $\texttt{Indef}(\beta') = \texttt{Indef}(\beta) \setminus \{\gamma\}$ and $f(\lambda_1) \neq \gamma$.

3. $\texttt{Def}(\beta) \subseteq range(f)$ by Definition 11
   $range(f') = range(f) \setminus \{\gamma\}$
   $\texttt{Def}(\beta') = \texttt{Def}(\beta) \Rightarrow \texttt{Def}(\beta') \subseteq range(f)$ since γ is not definite
   $\Rightarrow \texttt{Def}(\beta') \subseteq range(f')$ since $\gamma \notin \texttt{Def}(\beta')$.

By hypothesis, $\theta' \in Cc_i(\beta')$. By Definition 7, $\{\lambda_1, \ldots, \lambda_m\} \underline{\cup} \theta' \in Cc_i(\{\gamma\} \cup \beta')$.

Case γ is definite. By Definition 11, there exists a unique $\lambda_1 \in \theta$, such that $f(\lambda_1) = \gamma$. $\lambda_1 \in Cc(cons(\gamma))$ by Definition 11 and $\{\lambda_1\} \in Cc(\gamma)$ by Definition 6. Let $\theta = \{\lambda_1\} \underline{\cup} \theta'$ and consider the function $f' : \theta' \to \beta'$ defined by $f'(\lambda) = f(\lambda)$ for all $\lambda \in \theta'$. We show that f' is a lifting function of θ' to β'.

1. $\forall \lambda \in \theta',\ \lambda \in Cc(cons(f'(\lambda)))$ by Definition 11.

2. $\forall \lambda_1, \lambda_2 \in \theta', \lambda_1 \neq \lambda_2 \ \Rightarrow\ f(\lambda_1) \neq f(\lambda_2) \vee f(\lambda_1) \in \texttt{Indef}(\beta)$ by Definition 11
   $\forall \lambda_1, \lambda_2 \in \theta', \lambda_1 \neq \lambda_2 \ \Rightarrow\ f'(\lambda_1) \neq f'(\lambda_2) \vee f'(\lambda_1) \in \texttt{Indef}(\beta')$ since $\texttt{Indef}(\beta) = \texttt{Indef}(\beta')$.

3. $\texttt{Def}(\beta) \subseteq range(f)$ by Definition 11
   $range(f) = \{\gamma\} \cup range(f')$
   $\{\gamma\} \cup \texttt{Def}(\beta') = \texttt{Def}(\beta) \ \Rightarrow\ \texttt{Def}(\beta') \subseteq range(f')$ since γ is definite.

By hypothesis, $\theta' \in Cc_i(\beta')$. By Definition 7, $\{\lambda_1\} \underline{\cup} \theta' \in Cc_i(\{\gamma\} \cup \beta')$. □

Lemma 12 Let γ_1 and γ_2 be two abstract constraints with multiplicity and let $\gamma_1 \sqsubseteq \gamma_2$.
(i) If γ_2 is a definite constraint, then γ_1 is also a definite constraint.
(ii) If γ_1 is an indefinite constraint, then γ_2 is also an indefinite constraint.

Proof: Immediate consequence of Definitions 6 and 8. □

Theorem 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) $\beta_1 \sqsubseteq \beta_2 \Rightarrow Cc(\beta_1) \subseteq Cc(\beta_2)$.

Proof: (i) Let $\beta_1 \sqsubseteq \beta_2$ and $\theta \in Cc_i(\beta_1)$. We show that $\theta \in Cc_i(\beta_2)$. By Lemma 3, there exists a lifting function f of θ to β_1. By Definition 10 there exists an ordering function g of β_1 to β_2. Consider the function $g \circ f$. We show that $g \circ f$ is a lifting function of θ to β_2.

1. $\forall \lambda \in \theta : \lambda \in Cc(cons(f(\lambda)))$ by Definition 11
   $\forall f(\lambda) \in \beta_1 : f(\lambda) \sqsubseteq g(f(\lambda))$ by Definition 9
   $\forall f(\lambda) \in \beta_1 : cons(f(\lambda)) \sqsubseteq cons(g(f(\lambda)))$ by ordering on AM_D
   $\forall \lambda \in \theta : \lambda \in Cc(cons(g(f(\lambda))))$ by Lemma 1.

2. $\forall \lambda_1, \lambda_2 \in \theta : \lambda_1 \neq \lambda_2$
   $\Rightarrow f(\lambda_1) \neq f(\lambda_2) \vee f(\lambda_1) \in \texttt{Indef}(\beta_1)$ by Definition 11
   $\Rightarrow f(\lambda_1) \neq f(\lambda_2) \vee g(f(\lambda_1)) \in \texttt{Indef}(\beta_2)$ by $f(\lambda_1) \sqsubseteq g(f(\lambda_1))$ and Lemma 12
   $\Rightarrow g(f(\lambda_1)) \neq g(f(\lambda_2)) \vee g(f(\lambda_1)) \in \texttt{Indef}(\beta_2) \vee g(f(\lambda_1)) \in \texttt{Indef}(\beta_2)$ by Definition 9
   $\Rightarrow g(f(\lambda_1)) \neq g(f(\lambda_2)) \vee g(f(\lambda_1)) \in \texttt{Indef}(\beta_2)$

3. $\texttt{Def}(\beta_2) \subseteq range(g)$ by Definition 9
   $\Rightarrow \forall \gamma_2 \in \texttt{Def}(\beta_2)\ \exists \gamma_1 \in \beta_1 :\ g(\gamma_1) = \gamma_2$
   $\Rightarrow \forall \gamma_2 \in \texttt{Def}(\beta_2)\ \exists \gamma_1 \in \texttt{Def}(\beta_1) :\ g(\gamma_1) = \gamma_2$ $\gamma_1 \sqsubseteq g(\gamma_1) = \gamma_2$ and Lemma 12
   $\forall \gamma_1 \in \texttt{Def}(\beta_1)\ \exists \lambda \in \theta :\ f(\lambda) = \gamma_1$ by Definition 11
   $\Rightarrow \forall \gamma_2 \in \texttt{Def}(\beta_2)\ \exists \lambda \in \theta :\ g(f(\lambda)) = \gamma_2$ combining the above two statements
   $\Rightarrow \texttt{Def}(\beta_2) \subseteq range(g \circ f)$

$\theta \in Cc_i(\beta_2)$ follows by Lemma 3.
(ii) Now 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, $\theta \in Cc(\beta_2)$. □

Lemma 4 [Reduction of Ordering over AS_D to Weighted Bipartite Graph Matching] Let β_1 and β_2 be two abstract stores and G be a matching graph of β_1 to β_2. $\beta_1 \sqsubseteq \beta_2$ if and only if G has a matching of weight $|\beta_1| + |\texttt{Def}(\beta_2)|$.

Proof: ($\Rightarrow$) Let $\beta_1 \sqsubseteq \beta_2$. By Definition 10, 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,

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

We first prove that M is a matching. By Definition 9, for each edge in M, we have $\gamma_1 \sqsubseteq f(\gamma_1) = \gamma_2$. Therefore, $M_i \subseteq E_i \;\; (2 \leq i \leq 4)$, i.e., M is a set of edges of G. In addition,

1. each vertex γ_1 in V_1 appears in exactly one edge in M, since f is an ordering function (property 1);
2. each vertex γ_2 in V_2 appears in exactly one edge in M_2 by Definition 9 (properties 2 and 3);
3. each vertex γ_2 in V_3 appears in at most one edge in M_3 by Definition 9 (property 2);
4. each vertex $\gamma_2{}^{\gamma_1}$ in V_4 appears in at most one edge in M_4, by definition of M_4 since f is a function.

Hence, M is a matching of G.

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

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

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

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

1. Property 1 follows from the definition of the edges in the matching graph G;
2. Property 2 follows from the definition of a matching;

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

□

Addition

Theorem 5 Let β be an abstract store and γ be an abstract constraint with multiplicity.
(i) If $\theta_1 \in \mathit{Cc}_i(\beta)$ and $\theta_2 \in \mathit{Cc}(\gamma)$, then $\theta_1 \underline{\cup}\, \theta_2 \in \mathit{Cc}_i(\beta \uplus \gamma)$.
(ii) If $\theta_1 \in \mathit{Cc}(\beta)$ and $\theta_2 \in \mathit{Cc}(\gamma)$, then $\theta_1 \underline{\cup}\, \theta_2 \in \mathit{Cc}(\beta \uplus \gamma)$.

Proof: (i) Let $\gamma = \langle \sigma, \mu \rangle$. If $\theta_1 \in \mathit{Cc}_i(\beta)$, then there exists a lifting function f of θ_1 to β by Lemma 3. Consider the function $f' : \theta_1 \underline{\cup}\, \theta_2 \to \beta \uplus \gamma$ given by

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

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

1. $\forall \lambda_1 \in \theta_1 : \lambda_1 \in \mathit{Cc}(\mathit{cons}(f'(\lambda_1)))$ by Definition 11 and $f'(\lambda_1) = f(\lambda_1)$
$\forall \lambda_2 \in \theta_2 : \lambda_2 \in \mathit{Cc}(\mathit{cons}(f'(\lambda)))$ by Lemma 11, $\gamma \sqsubseteq f'(\lambda_2)$ and Lemma 1
$\Rightarrow\ \forall \lambda \in \theta_1 \underline{\cup}\, \theta_2 : \lambda \in \mathit{Cc}(\mathit{cons}(f'(\lambda)))$.

2. $\forall \lambda_1, \lambda_2 \in \theta_1 : \lambda_1 \neq \lambda_2$
$\Rightarrow f(\lambda_1) \neq f(\lambda_2) \vee f(\lambda_1) \in \mathtt{Indef}(\beta)$ by Definition 11
$\Rightarrow f'(\lambda_1) \neq f'(\lambda_2) \vee f'(\lambda_1) \in \mathtt{Indef}(\beta \uplus \gamma)$ since $f'(\lambda_1) = f(\lambda_1), f'(\lambda_2) = f(\lambda_2), \beta \subseteq \beta \uplus \gamma$

$\forall \lambda_1 \in \theta_1, \lambda_2 \in \theta_2 :$
$\gamma \notin \beta \ \Rightarrow f'(\lambda_1) \neq f'(\lambda_2)$ since $f'(\lambda_1) \in \beta, f'(\lambda_2) = \gamma$ and $\gamma \notin \beta$
$\gamma \in \beta \ \Rightarrow\ f'(\lambda_2) \in \mathtt{Indef}(\beta \uplus \gamma)$ since $\langle \sigma, \mathit{Any} \rangle$ is indefinite

$\forall \lambda_1, \lambda_2 \in \theta_2 : \lambda_1 \neq \lambda_2 \Rightarrow \gamma$ is indefinite by Definition 6.
$\Rightarrow \forall \lambda_1, \lambda_2 \in \theta_2 : \lambda_1 \neq \lambda_2 \Rightarrow f(\lambda_1) \in \mathtt{Indef}(\beta \uplus \gamma)$

3. $\mathtt{Def}(\beta) \subseteq \mathit{range}(f)$ By Definition 11
$\gamma \notin \beta \ \Rightarrow\ [\mathtt{Def}(\beta \uplus \gamma) \subseteq \{\gamma\} \cup \mathtt{Def}(\beta) \subseteq \{\gamma\} \cup \mathit{range}(f) = \mathit{range}(f')]$
$\gamma \in \beta \ \Rightarrow\ [\mathtt{Def}(\beta \uplus \gamma) = \mathtt{Def}(\beta) \subseteq \mathit{range}(f) \subseteq \mathit{range}(f')]$.

Therefore $\theta_1 \underline{\cup}\, \theta_2 \ \in \mathit{Cc}_i(\beta \ \uplus\ \gamma)$.
(ii) Consider now $\theta_1 \in \mathit{Cc}(\beta)$ and $\theta_2 \in \mathit{Cc}(\gamma)$. By definition of Cc, there exists $\theta_i \in \mathit{Cc}_i(\beta)$ s.t. $\theta_1 \leftrightarrow \theta_i$. By part (i), $\theta_i \underline{\cup}\, \theta_2 \ \in \mathit{Cc}_i(\beta \uplus \gamma)$. Since $\theta_1 \underline{\cup}\, \theta_2 \leftrightarrow \theta_i \underline{\cup}\, \theta_2$, by definition of $\mathit{Cc}\ \theta_1 \underline{\cup}\, \theta_2 \ \in \mathit{Cc}(\beta \uplus \gamma)$. □

Upper Bound

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

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

- *case* $2i$: We have that $\beta_1 = \{\gamma\} \cup \beta_1'$ $(\gamma \notin \beta_1')$ and $\gamma = \langle \sigma, One \rangle$. By definition of the concretization function, we have that $\theta = \theta_1 \underline{\cup} \theta_2$, $\theta_1 \in Cc(\gamma)$, and $\theta_2 \in Cc_i(\beta_1')$. By hypothesis, we have that $\theta_2 \in Cc_i(\texttt{UNION}(\beta_1', \beta_2))$. We also have that $\theta_1 \in Cc(\langle \sigma, ZeroOrOne \rangle)$ by monotonicity of Cc. Hence, by Theorem 5 (i), we have $\theta_1 \underline{\cup} \theta_2 \in Cc_i(\texttt{UNION}(\beta_1', \beta_2)) \uplus \langle \sigma, ZeroOrOne \rangle$.

- *case* $2ii$: By hypothesis, we have that $\theta \in Cc_i(\texttt{UNION}(\beta_1, \beta_2'))$. By definition of the concretization function, we have that $\emptyset \in Cc(\langle \sigma, ZeroOrOne \rangle)$. Hence, by Theorem 5 (i),
$$\theta \in Cc_i(\texttt{UNION}(\beta_1, \beta_2')) \uplus \langle \sigma, ZeroOrOne \rangle.$$

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

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

- *case* 4: We have that $\beta_1 = \{\gamma\} \cup \beta_1'$ $(\gamma \notin \beta_1')$ and $\gamma = \langle s_0 \ op \ \sum_{i=1}^n s_i x_i, \mu \rangle$. By definition of the concretization function, we have that $\theta = \theta_1 \underline{\cup} \theta_2$, $\theta_1 \in Cc(\gamma)$, and $\theta_2 \in Cc_i(\beta_1')$. By hypothesis, we have that $\theta_2 \in Cc_i(\texttt{UNION}(\beta_1', \beta_2'))$. By monotonicity of Cc, we have that $\theta_1 \in Cc(\langle (s_0 \sqcup s_0') \ op \ \sum_{i=1}^n (s_i \sqcup s_i') x_i, \mu \sqcup \mu' \rangle)$. The result follows from Theorem 5 (i).

(ii) Consider now $\theta \in Cc(\beta_1) \vee \theta \in Cc(\beta_2)$. By definition of Cc, there exists θ_i s.t. $\theta_i \in Cc_i(\beta_1) \vee \theta_i \in Cc_i(\beta_2)$ and $\theta \leftrightarrow \theta_i$. By part (i), $\theta_i \in Cc_i(\texttt{UNION}(\beta_1, \beta_2))$. By definition of Cc, $\theta \in Cc(\texttt{UNION}(\beta_1, \beta_2))$. □

Projection

Lemma 13 Let σ be an abstract constraint such that op(σ) is '=' and λ be a constraint.

$$\forall \lambda \in Cc(\sigma) : \texttt{Cneg}(\lambda) \in Cc(\texttt{Aneg}(\sigma)).$$

Proof: Direct consequence of the consistency of sign arithmetic. □

Lemma 14 Let $\sigma_1, \sigma_=^+$ be abstract constraints such that $\sigma_=^+[v] = \oplus \ \wedge$ op($\sigma_=^+$) is $'='$, $\lambda_1, \lambda_=^+$ be constraints and $v \in \aleph$.

$$\forall \lambda_1 \in Cc(\sigma_1), \lambda_=^+ \in Cc(\sigma_=^+) : \texttt{Celiminate}(\lambda_1, \lambda_=^+, v) \in Cc(\texttt{Aeliminate}(\sigma_1, \sigma_=^+, v)).$$

Proof: Direct consequence of the consistency of sign arithmetic. □

Lemma 15 Let σ^-, σ^+ be abstract constraints such that $\sigma^+[v] = \oplus \ \wedge \ \sigma^-[v] = \ominus$, λ^-, λ^+ be constraints and $v \in \aleph$.

$$\forall \lambda^- \in Cc(\sigma^-), \lambda^+ \in Cc(\sigma^+) : \texttt{Ccombine}(\lambda^-, \lambda^+, v) \in Cc(\texttt{Acombine}(\sigma^-, \sigma^+, v)).$$

Proof: Direct consequence of the consistency of sign arithmetic. □

Lemma 16 [Gauss] Let $v \in \aleph$, σ be an abstract constraint such that $\sigma[v] = \oplus$ and $op(\sigma)$ is '=' and β be an abstract store. We have

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

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

$$\begin{array}{lcl} \lambda & \in & Cc(\sigma) \\ \theta_1^i & \in & Cc(\gamma') \\ \theta_2^i & \in & Cc_i(\beta') \\ \theta_1 & = & \texttt{Cgauss}(\theta_1^i, \lambda, v) \\ \theta_2 & = & \texttt{Cgauss}(\theta_2^i, \lambda, v) \\ \gamma_1 & = & \langle \texttt{Aeliminate}(\sigma', \sigma, v), \mu' \rangle \\ \beta_2 & = & \texttt{Agauss}(\beta', \sigma, v) \end{array}$$

By hypothesis, we have

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

By Lemma 14, the definition of the concretization function for abstract constraints with multiplicity, and the fact that there are as many constraints in θ_1 as in θ_1^i, we have

$$\theta_1 \in Cc(\gamma_1).$$

The result follows from Theorem 5 (i). □

Lemma 17 [Split Top] Let θ be a constraint store, let β be an abstract store, and $v \in \aleph$. We have

$$\theta \in Cc_i(\beta) \Rightarrow \theta \in Cc_i(\texttt{Asplit_top}(\beta, v))$$

Proof: By induction on $|\beta|$. The basic case is trivial. Assume that the result holds for $|\beta| \leq n$. We show that it holds for $|\beta| = n+1$. Let $\beta = \{\gamma\} \cup \beta_2 \quad (\gamma = \langle \sigma, \mu \rangle \notin \beta_2)$ and $|\beta_2| = n$. Let $\theta \in Cc_i(\beta)$. Then, $\theta = \theta_1 \underline{\cup} \theta_2$ with $\theta_1 \in Cc(\gamma)$ & $\theta_2 \in Cc_i(\beta_2)$. By the induction hypothesis, we know that $\theta_2 \in Cc_i(\texttt{Asplit_top}(\beta_2, v))$. Now if $\sigma[v] \neq \top$, the result follows by Theorem 5 (i) . Otherwise θ_1 can be partitioned into three distinct sets θ_1^0, θ_1^+ and θ_1^-, depending on the coefficient of x_v. We have that $\theta_1^0 \in Cc(\gamma^0)$ since $\theta_1^0 \in Cc(\gamma)$, all constraints in θ_1^0 have a zero coefficient for x_v and $\mu \sqsubseteq \mu \sqcup ZeroOrOne$. Similarly $\theta_1^+ \in Cc(\gamma^+)$ and $\theta_1^- \in Cc(\gamma^-)$. By Theorem 5 (i), we have

$$\theta_1 = \theta_1^0 \underline{\cup} \theta_1^+ \underline{\cup} \theta_1^- \ \in Cc_i(\{\gamma^0, \gamma^+, \gamma^-\})$$

The result then follows from Lemma 5 (i). □

Lemma 18 [Split Basic] Let θ be a constraint store, let β be an abstract store, and $v \in \aleph$. Let $\sigma[v] \in \{0, \oplus, \ominus\}$ for all constraints $\langle \sigma, \mu \rangle \in \beta$. We have

$$\left.\begin{array}{l} \texttt{Csplit_basic}(\theta, v) = \langle \theta^0, \theta^+, \theta^- \rangle \\ \theta \in Cc_i(\beta) \\ \texttt{Asplit_basic}(\beta, v) = \langle \beta^0, \beta^+, \beta^- \rangle \end{array}\right\} \Rightarrow \theta^0 \in Cc_i(\beta^0) \wedge \theta^+ \in Cc_i(\beta^+) \wedge \theta^- \in Cc_i(\beta^-).$$

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

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

where

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

We show that $\theta^+ \in Cc_i(\beta^+)$. The other proofs are similar. Let $\gamma = \langle \sigma, \mu \rangle$.
Case $\sigma[v] = 0$. θ_1 does not contain any constraint λ such that $\lambda[v] = \oplus$ by definition of the concretization. Hence, $\theta^+ = \theta_2^+$ and the result follows from the fact that $\beta^+ = \beta_2^+$.
Case $\sigma[v] = \oplus$. θ_1 contains only constraints such that $\lambda[v] = \oplus$, $\theta^+ = \theta_2^+ \underline{\cup} \theta_1$ and $\beta^+ = \beta_2^+ \uplus \gamma$. The result follows by Theorem 5 (i).
Case $\sigma[v] = \ominus$ and $\mathrm{op}(\sigma)$ is '='. $\theta^+ = \theta_2^+$ and the result follows from the fact that $\beta^+ = \beta_2^+$.
Case $\sigma[v] = \ominus$ and $\mathrm{op}(\sigma)$ is '='. θ_1 contains only constraints such that $\lambda[v] = \ominus$, $\theta^+ = \theta_2^+ \underline{\cup} \theta_3$ where $\theta_3 = \{\texttt{Cneg}(\lambda) | \lambda \in \theta_1\}$ and $\beta^+ = \beta_2^+ \uplus \gamma_3$ where $\gamma_3 = \langle \texttt{Aneg}(\sigma), \mu \rangle$. Also, by Lemma 13, $\theta_1 \in Cc(\langle \sigma, \mu \rangle)$ and the fact that there are as many constraints in θ_3 as θ_1 , $\theta_3 \in Cc(\gamma_3)$. The result follows by Theorem 5 (i). □

Lemma 19 [Split] Let θ be a constraint store, let β be an abstract store, and $v \in \aleph$. We have

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

Proof: This follows from Lemma 17, the fact that for all $\gamma = \langle \sigma, \mu \rangle \in \texttt{Asplit_top}(\beta, v)$, $\sigma[v] \in \{0, \oplus, \ominus\}$ and the consistency of `Asplit_basic`. □

Lemma 20 [Fourier Step] Let $v \in \aleph$, let β^+ be an abstract store such that for all $\gamma \in \beta^+$, $\gamma = \langle \sigma, \mu \rangle$ and $\sigma[v] = \oplus$, let β^- be an abstract store such that for all $\gamma \in \beta^-$, $\gamma = \langle \sigma, \mu \rangle$ and $\sigma[v] = \ominus$. We have

$$\theta^+ \in Cc_i(\beta^+) \wedge \theta^- \in Cc_i(\beta^-) \Rightarrow \texttt{Cfourier_step}(\theta^+, \theta^-, v) \in Cc_i(\texttt{Afourier_step}(\beta^+, \beta^-, v)).$$

Proof: Let β^+ be $\{\gamma_1^+, \ldots, \gamma_n^+\}$ and β^- be $\{\gamma_1^-, \ldots, \gamma_m^-\}$. Let $\{(\gamma_1^+, \gamma_1^-), \ldots, (\gamma_n^+, \gamma_m^-)\}$ be the pairs considered in the algorithm. Consider the pair (γ_i^+, γ_j^-) $(1 \leq i \leq n \ \& \ 1 \leq$

$j \leq m$) such that $\gamma_i^+ = \langle \sigma_i^+ \ , \ \mu_i^+ \rangle$ and $\gamma_j^- = \langle \sigma_j^- \ , \ \mu_j^- \rangle$ and consider $\theta^+ \in Cc(\gamma_i^+)$ and $\theta^- \in Cc(\gamma_j^-)$. By Lemma 15, we have that any pair of constraints $\lambda^+ \in \theta^+$ and $\lambda^- \in \theta^-$ satisfies

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

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

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

The result follows from Theorem 5 (i). □

Lemma 21 [Fourier] Let $v \in \aleph$, θ be a store, and β be an abstract store. We have

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

Proof: This follows from the consistency of `Asplit` (Lemma 19) and of `Afourier_step` (Lemma 20) and from the definition of the concretization which specifies that, when γ is of the form $\langle \sigma, \mu \rangle$ with $\mu \in \{One, ZeroOrOne\}$, any $\theta \in Cc(\gamma)$ is of cardinality at most one. □

Theorem 7 [Project] Let $v \in \aleph$, θ be a store and β be an abstract store.

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

Proof: If $\beta = \emptyset$, the result is trivially true. Otherwise let β be $\{\gamma\} \cup \beta'$ ($\gamma \notin \beta'$) with $\gamma = \langle \sigma, \mu \rangle$ and $\theta \in Cc(\beta)$. By definition of Cc, there exists $\theta_i \in Cc_i(\beta)$ s.t. $\theta \leftrightarrow \theta_i$. If $\mu = One$, $\mathrm{op}(\sigma)$ is '=' and $\sigma[v] = \oplus$, then there exists a λ s.t. $\theta_i = \{\lambda\} \underline{\cup} \theta'$, $\lambda \in Cc(\gamma)$, and $\theta' \in Cc_i(\beta')$. By definition of the concretization, $\lambda[v] > 0$ and it follows from Lemma 16 that

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

and thus

$$\texttt{Cproject}(\theta_i, v) \in Cc_i(\texttt{Aproject}(\beta, v))$$

Since $\texttt{Cproject}(\theta, v) \leftrightarrow \texttt{Cproject}(\theta_i, v)$, it follows by definition of Cc that

$$\texttt{Cproject}(\theta, v) \in Cc(\texttt{Aproject}(\beta, v))$$

The case $\mu = One$, $\mathrm{op}(\sigma)$ is '=' and $\sigma[v] = \ominus$ is similar. In the last case, we have by Lemma 21 that

$$\texttt{Cfourier}(\theta_i, v) \in Cc_i(\texttt{Afourier}(\beta, v))$$

Moreover, since $\texttt{Cproject}(\theta_i, v) \leftrightarrow \texttt{Cfourier}(\theta_i, v)$, and $\texttt{Cproject}(\theta, v) \leftrightarrow \texttt{Cproject}(\theta_i, v)$, it follows by definition of Cc that

$$\texttt{Cproject}(\theta, v) \in Cc(\texttt{Aproject}(\beta, v))$$

□

Satisfiability

Lemma 22 [Is Trivially Satisfiable] Let θ be a store and β be an abstract store.
(i) $\theta \in Cc_i(\beta) \Rightarrow (\texttt{Ais_triv_sat}(\beta) \Rightarrow \texttt{Cis_triv_sat}(\theta))$
(ii) $\theta \in Cc(\beta) \wedge \texttt{Var}(\theta) = \emptyset \wedge \texttt{Var}(\beta) = \emptyset$
$\Rightarrow (\texttt{Ais_triv_sat}(\beta) \Rightarrow \texttt{Cis_triv_sat}(\theta))$

Proof: (i) By induction on $|\beta|$. The basic case is obvious. Assume that the result holds for $|\beta| \leq n$. We show that it holds for $|\beta| = n+1$. Let $\beta = \{\gamma\} \cup \beta' \quad (\gamma \notin \beta')$ with $\gamma = \langle \sigma, \mu \rangle$ and $|\beta'| = n$. Let $\theta \in Cc_i(\beta)$. Then, $\theta = \theta_1 \bigsqcup \theta_2$ with $\theta_1 \in Cc(\gamma)$ & $\theta_2 \in Cc_i(\beta')$.
Case σ is $s_0 < \sum_{i=1}^{n} 0x_i \ \wedge \ s_0 = \ominus$. By definition of the concretization for abstract constraints,

$$\forall \lambda \in \theta_1 : \lambda \text{ is } c_0 < \sum_{i=1}^{n} 0x_i \ \wedge \ c_0 < 0$$

Therefore $\texttt{C_is_triv_sat}(\theta_1)$ is true. This gives

$$\texttt{C_is_triv_sat}(\theta) = \texttt{C_is_triv_sat}(\theta_1) \ \wedge \ \texttt{C_is_triv_sat}(\theta_2) = \texttt{C_is_triv_sat}(\theta_2)$$

But $\texttt{A_is_triv_sat}(\beta) = \texttt{A_is_triv_sat}(\beta')$. The desired result follows from the hypothesis.
Case σ is $s_0 = \sum_{i=1}^{n} 0x_i \ \wedge \ s_0 = 0$. The arguments are similar.
Case σ is $s_0 \leq \sum_{i=1}^{n} 0x_i \ \wedge \ (s_0 = 0 \vee s_0 = \ominus)$. The arguments are similar.
Otherwise. There is nothing to prove as $\texttt{A_is_triv_sat}(\beta)$ is false.
(ii) Let $\theta \in Cc(\beta)$. By definition of Cc, there exists $\theta_i \in Cc_i(\beta)$ s.t. $\theta \leftrightarrow \theta_i$. The precondition $\texttt{Var}(\beta) = \emptyset$ gives $\texttt{Var}(\theta_i) = \emptyset$. Moreover $\texttt{Var}(\theta) = \emptyset$. This implies that $\texttt{C_is_triv_sat}(\theta) = \texttt{C_is_triv_sat}(\theta_i)$. The desired result then follows from part (i). □

Theorem 8 [Is Satisfiable] Let θ be a store and β be an abstract store.

$$\theta \in Cc(\beta) \Rightarrow (\texttt{Ais_sat}(\beta) \Rightarrow \texttt{Cis_sat}(\theta))$$

Proof: Let $\theta \in Cc(\beta)$. By Corollary 1, we have that $\theta_p \in Cc(\beta_p)$. Also, $\texttt{Var}(\beta_p) = \emptyset$ and $\texttt{Var}(\theta_p) = \emptyset$. The desired result follows by Lemma 22 (ii). □

Unsatisfiability

Lemma 23 [Is Trivially Unsatisfiable] Let θ be a store and β be an abstract store.
(i) $\theta \in Cc_i(\beta) \Rightarrow (\texttt{Ais_triv_unsat}(\beta) \Rightarrow \texttt{Cis_triv_unsat}(\theta))$
(ii) $\theta \in Cc(\beta) \wedge \texttt{Var}(\theta) = \emptyset \wedge \texttt{Var}(\beta) = \emptyset$
$\Rightarrow (\texttt{Ais_triv_unsat}(\beta) \Rightarrow \texttt{Cis_triv_unsat}(\theta))$

Proof: (i) By induction on $|\beta|$. The basic case is obvious. Assume that the result holds for $|\beta| \leq n$. We show that it holds for $|\beta| = n+1$. Let $\beta = \{\gamma\} \cup \beta' \quad (\gamma \notin \beta')$ with $\gamma = \langle \sigma, \mu \rangle$ and $|\beta'| = n$. Let $\theta \in Cc_i(\beta)$. Then, $\theta = \theta_1 \bigsqcup \theta_2$ with $\theta_1 \in Cc(\gamma)$ & $\theta_2 \in Cc_i(\beta')$.
Case σ is $s_0 < \sum_{i=1}^{n} 0x_i \ \wedge \ (s_0 = 0 \vee s_0 = \oplus) \ \wedge \ \mu = One$. We have that $\texttt{Ais_triv_unsat}(\beta)$ is true. By definition of the concretization for abstract constraints,

$$\exists \lambda \in \theta : \lambda \text{ is } c_0 < \sum_{i=1}^{n} 0x_i \ \wedge \ c_0 \geq 0$$

Therefore $\texttt{C_is_triv_unsat}(\theta_1)$ is true. The desired result follows.
Case σ is $s_0 = \sum_{i=1}^{n} 0x_i \ \wedge \ (s_0 = \oplus \vee s_0 = \ominus)$. The arguments are similar.
Case σ is $s_0 \leq \sum_{i=1}^{n} 0x_i \ \wedge \ s_0 = \oplus$. The arguments are similar.
Otherwise. $\texttt{A_is_triv_unsat}(\beta) = \texttt{Ais_triv_unsat}(\beta')$. If $\texttt{Cis_triv_unsat}(\theta)$ is true, there is nothing to prove. Otherwise $\texttt{Cis_triv_unsat}(\theta) = \texttt{Cis_triv_unsat}(\theta')$, and the result follows by the induction hypothesis.
(ii) Let $\theta \in Cc(\beta)$. By definition of Cc, there exists $\theta_i \in Cc_i(\beta)$ s.t. $\theta \leftrightarrow \theta_i$. The precondition $\texttt{Var}(\beta) = \emptyset$ gives $\texttt{Var}(\theta_i) = \emptyset$. Moreover $\texttt{Var}(\theta) = \emptyset$. This implies that $\texttt{C_is_triv_unsat}(\theta) = \texttt{C_is_triv_unsat}(\theta_i)$. The desired result then follows from part (i). □

Theorem 9 [Is Unsatisfiable] Let θ be a store and β be an abstract store.

$$\theta \in Cc(\beta) \Rightarrow (\texttt{Ais_unsat}(\beta) \Rightarrow \texttt{Cis_unsat}(\theta))$$

Proof: Let $\theta \in Cc(\beta)$. By Corollary 1, we have that $\theta_p \in Cc(\beta_p)$. Also, $\texttt{Var}(\beta_p) = \emptyset$ and $\texttt{Var}(\theta_p) = \emptyset$. The desired result follows by Lemma 23 (ii). □

Conditional Satisfiability

Lemma 24 [Reduce Grounds] Let θ be a store and β be an abstract store.
(i)$\theta \in Cc_i(\beta) \ \Rightarrow \ \texttt{Cred_gnd}(\theta) \in Cc_i(\texttt{Ared_gnd}(\beta))$.
(ii)$\theta \in Cc(\beta) \ \wedge \ \theta$ satisfiable $\Rightarrow \ \texttt{Cred_gnd}(\theta) \in Cc(\texttt{Ared_gnd}(\beta))$.
(iii)$\theta \in Cc(\beta) \ \wedge \ \theta$ satisfiable $\Rightarrow \ \theta \in Cc(\texttt{Ared_gnd}(\beta))$.

Proof: (i) By induction on $|\beta|$. The basic case is obvious. Assume that the result holds for $|\beta| \leq n$. We show that it holds for $|\beta| = n + 1$. Let β be $\{\gamma\} \cup \beta' \quad (\gamma \notin \beta')$ with $\gamma = \langle \sigma, \mu \rangle$ and $|\beta'| = n$. Let $\theta \in Cc_i(\beta)$. Then $\theta = \theta_1 \underline{\cup} \theta_2$ with $\theta_1 \in Cc(\gamma)$ and $\theta_2 \in Cc_i(\beta')$. According to the induction hypothesis, we have

$$\texttt{Cred_gnd}(\theta_2) \in Cc_i(\texttt{Ared_gnd}(\beta')).$$

Case σ is $s_o \ op \ \sum_{i=1}^{n} 0x_i$. By definition of the concretization function for abstract constraints with multiplicity, we have that θ_1^i consists only of constraints of the form $c_0 \ op \ \sum_{i=1}^{n} 0x_i$ and therefore $\texttt{Cred_gnd}(\theta_1) = \emptyset$. This gives

$$\texttt{Cred_gnd}(\theta) = \texttt{Cred_gnd}(\theta_1) \underline{\cup} \texttt{Cred_gnd}(\theta_2) = \texttt{Cred_gnd}(\theta_2).$$

But $\texttt{Ared_gnd}(\beta) = \texttt{Ared_gnd}(\beta')$. The desired result follows from the induction hypothesis.
Case σ is not $s_o \ op \ \sum_{i=1}^{n} 0x_i$. Again by definition of the concretization function for abstract constraints with multiplicity, we have that $\texttt{Cred_gnd}(\theta_1) = \theta_1$. This gives $\texttt{Cred_gnd}(\theta) = \texttt{Cred_gnd}(\theta_2) \underline{\cup} \theta_1$. But $\texttt{Ared_gnd}(\beta) = \texttt{Ared_gnd}(\beta') \uplus \gamma$. The desired result follows from the induction hypothesis and Theorem 5 (i).
(ii) Let $\theta \in Cc(\beta)$. By definition of Cc, there exists $\theta_i \in Cc_i(\beta)$ s.t. $\theta \leftrightarrow \theta_i$. By part (i) we have that

$$\texttt{Cred_gnd}(\theta_i) \in Cc_i(\texttt{Ared_gnd}(\beta)).$$

But θ satisfiable implies that $\texttt{Cred_gnd}(\theta) \leftrightarrow \texttt{Cred_gnd}(\theta_i)$. The desired result follows by the definition of Cc.
(iii) This follows from part (ii), the definition of Cc, and the fact that if θ is satisfiable, then $\theta \leftrightarrow \texttt{Cred_gnd}(\theta)$. □

Lemma 25 [Reduce Equations Step] Let $v \in \aleph$, θ be a store and β be an abstract store.
(i)$\theta \in Cc(\beta) \Rightarrow \texttt{Cred_eqn_step}(\theta, v) \in Cc(\texttt{Ared_eqn_step}(\beta, v))$.
(ii)$\theta \in Cc(\beta) \Rightarrow \theta \in Cc(\texttt{Ared_eqn_step}(\beta, v))$.

Proof: (i) If $\beta = \emptyset$ the result is obvious. Otherwise, let $\beta = \{\gamma\} \cup \beta' \quad (\gamma \notin \beta')$ with $\gamma = \langle \sigma, \mu \rangle$. Let $\theta \in Cc(\beta)$. Then, by definition of Cc, there exists $\theta_i \in Cc_i(\beta)$ s.t. $\theta \leftrightarrow \theta_i$. Case $\mu = One \wedge \text{op}(\sigma)$ is '=' $\wedge\ \sigma[v] = \oplus$. There exists a λ s.t. $\theta_i = \{\lambda\} \underline{\cup} \theta'$, $\lambda \in Cc(\sigma)$, and $\theta' \in Cc_i(\beta')$. By definition of the concretization, $\lambda[v] > 0$ and it follows from Lemma 16 that

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

and thus by definition of Cc that

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

It follows from $\lambda \in Cc(\gamma)$ and Theorem 5 (ii) that

$$\texttt{Cred_eqn_step}(\theta, v) \in Cc(\texttt{Ared_eqn_step}(\beta, v)).$$

Case $\mu = One \wedge \text{op}(\sigma)$ is '=' $\wedge\ \sigma[v] = \ominus$. The arguments are very similar.
Otherwise. We have that $\texttt{Ared_eqn_step}(\beta, v) = \beta$. Also we know that $\theta \leftrightarrow \texttt{Cred_eqn_step}(\theta, v)$. The desired result follows by the definition of Cc for abstract stores.
(ii) This follows from part (i), the definition of Cc and the fact that $\theta \leftrightarrow \texttt{Cred_eqn_step}(\theta, v)$.
□

Lemma 26 [Reduce Equations Set] Let $V \in 2^{\aleph}$, θ be a store and β be an abstract store.

$$\theta \in Cc(\beta) \Rightarrow \theta \in Cc(\texttt{Ared_eqn_set}(\beta, V)).$$

Proof: The proof is a simple induction of applying Lemma 25 (ii) $|V|$ times. □

Lemma 27 [Reduce Equations] Let θ be a store and β be an abstract store.

$$\theta \in Cc(\beta) \Rightarrow \theta \in Cc(\texttt{Ared_eqn}(\beta)).$$

Proof: Direct consequence of Lemma 26. □

Lemma 28 [Reduce Tops] Let θ be a store and β be an abstract store.

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

Proof: If $\beta = \emptyset$ or $|\beta| = 1$, the result is obvious. Otherwise, let $\beta = \{\gamma_1, \gamma_2\} \cup \beta' \quad (\gamma_1 \notin \beta',\ \gamma_2 \notin \beta')$ with $\gamma_1 = \langle \sigma_1, \mu_1 \rangle$ and $\gamma_1 = \langle \sigma_2, \mu_2 \rangle$. Let $\theta \in Cc(\beta)$. Then, by definition of Cc, there exists $\theta_i \in Cc_i(\beta)$ s.t. $\theta \leftrightarrow \theta_i$. We consider one subcase of the first case statement in the algorithm.

Case $\mu = One \ \wedge (\sigma_1 \text{ is } \oplus \leq \oplus x_v) \ \wedge (\sigma_2 \text{ is } \top = \oplus x_v)$. There exists a λ_1 s.t. $\theta_i = \{\lambda_1\} \underline{\cup} \theta_2 \underline{\cup} \theta'$, $\lambda_1 \in Cc(\sigma_1)$, $\theta_2 \in Cc(\gamma_2)$ and $\theta' \in Cc_i(\beta')$. By definition of the concretization, λ_1 is of the form $c_0 \leq c_v x_v$ where $c_0 > 0$ and $c_v > 0$. Consider any $\lambda_2 \in \theta_2$. If λ_2 is of the form $c_0 = c_v x_v$ where $c_0 \leq 0$ and $c_v > 0$, then $\{\lambda_1\} \underline{\cup} \theta_2$ is unsatisfiable. So for all $\lambda_2 \in \theta_2$, λ_2 is of the form $c_0 = c_v x_v$ where $c_0 > 0$ and $c_v > 0$. By definition of the oncretization for abstract constraints, $\theta_2 \in Cc(\langle \oplus = \oplus x_v, \mu_2 \rangle)$. By the induction hypothesis and Lemma 5 (i) it follows that $\theta_i \in Cc_i(\texttt{Ared_top}(\beta))$. The desired result follows by the definition of Cc.
Other subcases and second case. Proof is similar.
Otherwise. We have that $\texttt{Ared_top}(\beta) = \beta$ and so there is nothing to prove. □

Theorem 10 [Reduce] Let θ be a store and β be an abstract store. Then

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

Proof: Consequence of Lemma 28, Lemma 27, and Lemma 24 (iii) □

Theorem 11 [Conditional Satisfiability] Let θ_1, θ_2 be stores and β_1, β_2 be abstract stores. Then

$$\theta_1 \in Cc(\beta_1) \ \wedge \ \theta_2 \in Cc(\beta_2) \Rightarrow (\texttt{Ais_cond_sat}(\beta_1, \beta_2) \Rightarrow \texttt{Cis_cond_sat}(\theta_1 \underline{\cup} \theta_2))$$

Proof:

$\texttt{Ais_cond_sat}(\beta_1, \beta_2)$
$= \texttt{Ais_unsat}(\beta_1) \ \vee \ \texttt{Ais_sat}(\texttt{Areduce}(\texttt{Aproject_set}(\beta_1, \texttt{Var}(\beta_1) \setminus \texttt{Var}(\beta_2))) \uplus \beta_2)$.

And its concrete counterpart

$$\texttt{Cis_cond_sat}(\theta_1, \theta_2) = \texttt{Cis_unsat}(\theta_1) \ \vee \ \texttt{Cis_sat}(\theta_1 \underline{\cup} \theta_2).$$

Let $V = \texttt{Var}(\beta_1) \setminus \texttt{Var}(\beta_2)$. For all variables $v \in V$, we have that $v \notin \texttt{Var}(\theta_2)$. Therefore

$\texttt{Cis_sat}(\theta_1 \underline{\cup} \theta_2)$
$= \texttt{Cis_triv_sat}(\texttt{Cproject_set}(\theta_1 \underline{\cup} \theta_2, \aleph))$
$= \texttt{Cis_triv_sat}(\texttt{Cproject_set}(\texttt{Cproject_set}(\theta_1, V) \underline{\cup} \theta_2, \aleph))$
$= \texttt{Cis_sat}(\texttt{Cproject_set}(\theta_1, V) \underline{\cup} \theta_2)$

By Corollary 1 and Theorem 10, we have that

$$\texttt{Cproject_set}(\theta_1, V) \in Cc(\texttt{Areduce}(\texttt{Aproject_set}(\beta_1, V))).$$

The desired result then follows from Lemma 5 and Theorems 8 and 9. □

Widening

Before proving the correctness of the widening, we need to introduce some additional notation. We introduce the functions `n_infty` and `n_zero` which give the number of infinity and zero symbols respectively in an abstract object. For example, $\texttt{n_infty}(\alpha)$ denotes the number of infinity symbols in the abstract multistore α. We also introduce

the function $\texttt{n_infty_zero}$ over abstract objects which gives the ordered pair whose first element is the number of infinity symbols and second element is the number of zero symbols in an abstract object, i.e. $\texttt{n_infty_zero}(\alpha) = \langle \texttt{n_infty}(\alpha), \texttt{n_zero}(\alpha) \rangle$. For each n, we consider the set of normalized abstract multistores χ_n over the variables $\{x_1, \ldots, x_n\}$. For each normalized abstract multistore α, we define its size denoted as $|\alpha|$ which is the number of abstract stores in α. Now, there are $3 \times 4^{n+1}$ abstract constraints with different shapes over the variables $\{x_1, \ldots, x_n\}$ (corresponding to three multiplicities and 4 ways to assign the signs of the constant and n coefficients). A normalized abstract store can have at most $3 \times 4^{n+1}$ constraints as all the constraints are required to have different shapes. As normalized abstract stores have constraints with all different shapes, there can be at most $2^{3 \times 4^{n+1}}$ normalized abstract stores with different shapes. A normalized abstract multistore contains normalized abstract stores with all different shapes. Therefore, for any normalized abstract multistore α, we have that

$$0 \leq |\alpha| \leq 2^{3 \times 4^{n+1}}$$

Further, if α is a normalized abstract multistore, each abstract store in α has at most $3 \times 4^{n+1}$ constraints which corresponds to $3 \times 4^{n+1} \times (n+1)$ infinity or zero symbols. Therefore

$$0 \leq \texttt{n_infty_zero}(\alpha) \leq 2^{3 \times 4^{n+1}} \times (3 \times 4^{n+1} \times (n+1))^2$$

though the upper bound is over estimated. For each normalized abstract multistore, we introduce its $\texttt{measure}$ defined as

$$\texttt{measure}(\alpha) = \langle |\alpha|, \texttt{n_infty_zero}(\alpha) \rangle$$

The set

$$\texttt{Measure}_n = \{\texttt{measure}(\alpha) | \alpha \in \chi_n\}$$

is finite (because of the bounds on $|\alpha|$ and $\texttt{n_infty_zero}(\alpha)$) and can be totally ordered by

$$\begin{array}{l} \texttt{measure}(\alpha_1) < \texttt{measure}(\alpha_2) \\ \Leftrightarrow (|\alpha_1| < |\alpha_2|) \ \vee \ ((|\alpha_1| = |\alpha_2|) \wedge (\texttt{n_infty_zero}(\alpha_1) < \texttt{n_infty_zero}(\alpha_2))) \end{array}$$

where

$$\begin{array}{l} \texttt{n_infty_zero}(\alpha_1) < \texttt{n_infty_zero}(\alpha_2) \\ \Leftrightarrow (\texttt{n_infty}(\alpha_1) < \texttt{n_infty}(\alpha_2)) \vee \\ \quad ((\texttt{n_infty}(\alpha_1) = \texttt{n_infty}(\alpha_2)) \wedge (\texttt{n_zero}(\alpha_1) < \texttt{n_zero}(\alpha_2))) \end{array}$$

.

Lemma 6 Let s_{old} and s_{new} be intervals s.t. $\texttt{shape}(s_{new}) = \texttt{shape}(s_{old})$. Then
(i) $\texttt{shape}(s_{old}) = \texttt{shape}(s_{new} \nabla s_{old})$.
(ii) $c \in Cc(s_{old}) \vee c \in Cc(s_{new}) \Rightarrow c \in Cc(s_{new} \nabla s_{old})$.
(iii) $\texttt{n_infty_zero}(s_{old}) \leq \texttt{n_infty_zero}(s_{new} \nabla s_{old})$.
(iv) $s_{new} \not\sqsubseteq s_{old} \Rightarrow \texttt{n_infty_zero}(s_{old}) < \texttt{n_infty_zero}(s_{new} \nabla s_{old})$.

Proof: (i) If $\texttt{pos}(s_{old})$. $\texttt{left}(s_{new} \nabla s_{old}) = \texttt{left}(s_{old})$ and $\texttt{open_left}(s_{new} \nabla s_{old}) = \texttt{open_left}(s_{old})$ or $\texttt{left}(s_{new} \nabla s_{old}) = 0$ and $\texttt{open_left}(s_{new} \nabla s_{old}) = \texttt{true}$. In either case, we have that $\texttt{pos}(s_{new} \nabla s_{old})$.

If $\texttt{neg}(s_{old})$. The arguments are similar about the right endpoint.
If $\texttt{zer}(s_{old})$. $\texttt{zer}(s_{new})$ and $s_{new} \nabla s_{old} = s_{old}$.
Otherwise. $\texttt{shape}(s_{old}) = \top$. We have that $\texttt{left}(s_{new} \nabla s_{old}) = \texttt{left}(s_{old}) \leq 0$ and $\texttt{open_left}(s_{new} \nabla s_{old}) = \texttt{open_left}(s_{old})$ or $\texttt{left}(s_{new} \nabla s_{old}) = -\infty$ and $\texttt{open_left}(s_{new} \nabla s_{old}) = \texttt{true}$. This implies $\neg\texttt{pos}(s_{new} \nabla s_{old})$. It can be similarly proved that $\neg\texttt{neg}(s_{new} \nabla s_{old})$, by considering the right endpoint. It can also be proved that $\neg\texttt{zer}(s_{new} \nabla s_{old})$ by considering both endpoints. This gives $\texttt{shape}(s_{new} \nabla s_{old}) = \top$.
(ii) If $\neg\texttt{decrease}(s_{new}, s_{old})$. We have that the left limit of $s_{new} \nabla s_{old}$ is the same as that of s_{old} and therefore $\neg\texttt{decrease}(s_{old}, s_{new} \nabla s_{old})$ and $\neg\texttt{decrease}(s_{new}, s_{new} \nabla s_{old})$.
If $\texttt{decrease}(s_{new}, s_{old}) \wedge \texttt{pos}(s_{old})$. The left limit of $s_{new} \nabla s_{old}$ is the minimum possible for any positive interval and therefore $\neg\texttt{decrease}(s_{old}, s_{new} \nabla s_{old})$ and $\neg\texttt{decrease}(s_{new}, s_{new} \nabla s_{old})$.
If $\texttt{decrease}(s_{new}, s_{old}) \wedge \neg\texttt{pos}(s_{old})$. The left limit of $s_{new} \nabla s_{old}$ is the minimum possible for any interval and therefore $\neg\texttt{decrease}(s_{old}, s_{new} \nabla s_{old})$ and $\neg\texttt{decrease}(s_{new}, s_{new} \nabla s_{old})$.
In all cases, we have that $\neg\texttt{decrease}(s_{old}, s_{new} \nabla s_{old})$ and $\neg\texttt{decrease}(s_{new}, s_{new} \nabla s_{old})$. Considering the right endpoint, we can similarly prove that $\neg\texttt{increase}(s_{old}, s_{new} \nabla s_{old})$ and $\neg\texttt{increase}(s_{new}, s_{new} \nabla s_{old})$.

From the definition, we have that $s_{old} \sqsubseteq s_{new} \nabla s_{old}$ and $s_{new} \sqsubseteq s_{new} \nabla s_{old}$. The desired result follows by monotonicity of the concretization over intervals.
(iii) The left endpoint of an interval can remain unchanged (if $\neg\texttt{decrease}(s_{new}, s_{old})$) or become an infinity symbol (if $\texttt{decrease}(s_{new}, s_{old}) \wedge \neg\texttt{pos}(s_{old})$) or become a zero (if $\texttt{decrease}(s_{new}, s_{old}) \wedge \texttt{pos}(s_{old})$). Also in the last case, the left endpoint cannot be originally an infinity symbol because $\texttt{pos}(s_{old})$. Similar arguments apply at the right endpoint of an interval. The desired result follows.
(iv) By definition of the ordering on intervals,

$$\neg(\neg\texttt{increase}(s_{new}, s_{old}) \wedge \neg\texttt{decrease}(s_{new}, s_{old}))$$

i.e.

$$\texttt{increase}(s_{new}, s_{old}) \vee \texttt{decrease}(s_{new}, s_{old}).$$

If $\texttt{decrease}(s_{new}, s_{old})$, the left endpoint of the interval can become an infinity symbol (if $\neg\texttt{pos}(s_{old})$) or become a zero (if $\texttt{pos}(s_{old})$). Also in the last case, the left endpoint cannot be originally an infinity symbol because $\texttt{pos}(s_{old})$. Similar arguments apply if $\texttt{increase}(s_{new}, s_{old})$. In conjunction with part (iii), the desired result follows. □

Lemma 7 Let γ_{old} and γ_{new} be abstract constraints with multiplicity s.t. $\texttt{shape}(\gamma_{new}) = \texttt{shape}(\gamma_{old})$. Then
(i) $\texttt{shape}(\gamma_{old}) = \texttt{shape}(\gamma_{new} \nabla \gamma_{old})$.
(ii) $\theta \in Cc(\gamma_{old}) \vee \theta \in Cc(\gamma_{new}) \Rightarrow \theta \in Cc(\gamma_{new} \nabla \gamma_{old})$.
(iii) $\texttt{n_infty_zero}(\gamma_{old}) \leq \texttt{n_infty_zero}(\gamma_{new} \nabla \gamma_{old})$.
(iv) $\gamma_{new} \not\sqsubseteq \gamma_{old} \Rightarrow \texttt{n_infty_zero}(\gamma_{old}) < \texttt{n_infty_zero}(\gamma_{new} \nabla \gamma_{old})$.

Proof: (i) This is a direct consequence of Lemma 6 (i).
(ii) This is a direct consequence of Lemma 6 (ii) and the definition of the concretization of an abstract constraint with multiplicity.
(iii) Direct consequence of Lemma 6 (iii).

(iv) Let $\gamma_{new} = \langle s_0^{new} \ \delta \ \sum_{i=1}^n s_i^{new} x_i, \mu \rangle$ and $\gamma_{old} = \langle s_0^{old} \ \delta \ \sum_{i=1}^n s_i^{old} x_i, \mu \rangle$. By definition of the ordering on abstract constraints, if $\gamma_{new} \not\sqsubseteq \gamma_{old}$ then

$$\exists j : s_j^{new} \not\sqsubseteq s_j^{old} \quad (0 \leq j \leq n)$$

Applying Lemma 6 (iv) in conjunction with part (iii) leads to the desired result. □

Lemma 8 Let β_{old} and β_{new} be normalized abstract stores s.t. $\texttt{shape}(\beta_{new}) = \texttt{shape}(\beta_{old})$. Then

(i) $\texttt{shape}(\beta_{old}) = \texttt{shape}(\beta_{new} \nabla \beta_{old})$ and $\beta_{new} \nabla \beta_{old}$ is normalized.
(ii) $\theta \in Cc(\beta_{old}) \vee \theta \in Cc(\beta_{new}) \Rightarrow \theta \in Cc(\beta_{new} \nabla \beta_{old})$.
(iii) $\texttt{n_infty_zero}(\beta_{old}) \leq \texttt{n_infty_zero}(\beta_{new} \nabla \beta_{old})$.
(iv) $\beta_{new} \not\sqsubseteq \beta_{old} \Rightarrow \texttt{n_infty_zero}(\beta_{old}) < \texttt{n_infty_zero}(\beta_{new} \nabla \beta_{old})$.

Proof: (i) The result is a direct consequence of Lemma 7 (i) and the one to one correspondence between $\gamma_{old} \in \beta_{old}$ and $\gamma_{new} \nabla \gamma_{old} \in \beta_{new} \nabla \beta_{old}$.
(ii) The basic case i.e. $|\beta_{new}| = |\beta_{old}| = 0$, is trivial. Assume the result for n. We prove it for $n+1$. We have

$$\begin{array}{l} \beta_{new} = \{\gamma_{new}\} \cup \beta_1, \quad \gamma_{new} \notin \beta_1, \ |\beta_1| = n \\ \beta_{old} = \{\gamma_{old}\} \cup \beta_2, \quad \gamma_{old} \notin \beta_2, \ |\beta_2| = n \\ \texttt{shape}(\beta_1) = \texttt{shape}(\beta_2) \\ \beta_{new} \nabla \beta_{old} = (\beta_1 \nabla \beta_2) \ \cup \ (\gamma_{new} \nabla \gamma_{old}) = (\beta_1 \nabla \beta_2) \uplus (\gamma_{new} \nabla \gamma_{old}) \\ \texttt{shape}(\gamma_{new}) = \texttt{shape}(\gamma_{old}) \end{array}$$

Let $\theta \in Cc(\beta_{new})$. Therefore by definition of Cc, $\theta = \theta_1 \underline{\cup} \theta_2$, where

$$\theta_1 \in Cc(\beta_1) \wedge \theta_2 \in Cc(\gamma_{new}).$$

By the induction hypothesis and Lemma 7 (ii)

$$\theta_1 \in Cc(\beta_1 \nabla \beta_2) \wedge \theta_2 \in Cc(\gamma_{new} \nabla \gamma_{old}).$$

It follows by Theorem 5 that

$$\theta \in Cc(\beta_{new} \nabla \beta_{old}).$$

This completes the proof for β_{new}. The proof for β_{old} is very similar.
(iii) The result is a direct consequence of Lemma 7 (iii) and the one to one correspondence between $\gamma_{old} \in \beta_{old}$ and $\gamma_{new} \nabla \gamma_{old} \in \beta_{new} \nabla \beta_{old}$.
(iv) If $(\beta_{new} \not\sqsubseteq \beta_{old})$, then

$$\exists \gamma_{old} \in \beta_{old}, \gamma_{new} \in \beta_{new} : \ \texttt{shape}(\gamma_{old}) = \texttt{shape}(\gamma_{new}) \ \wedge \ \gamma_{new} \not\sqsubseteq \gamma_{old}.$$

Applying Lemma 7 (iv) in conjunction with part (iii) leads to the desired result. □

Lemma 9 Let α_{old} and α_{new} be normalized abstract multistores. Then

(i) $\alpha_n \nabla \alpha_{old}$ is normalized.
(ii) $\theta \in Cc(\alpha_{old}) \vee \theta \in Cc(\alpha_{new}) \Rightarrow \theta \in Cc(\alpha_{new} \nabla \alpha_{old})$.
(iii) $|\alpha_{old}| \leq |\alpha_{new} \nabla \alpha_{old}|$.
(iv) $(\alpha_{new} \not\sqsubseteq \alpha_{old}) \ \wedge \ (|\alpha_{old}| = |\alpha_{new} \nabla \alpha_{old}|)$
$\Rightarrow \texttt{n_infty_zero}(\alpha_{old}) < \texttt{n_infty_zero}(\alpha_{new} \nabla \alpha_{old})$

Proof: (i) Direct consequence of the definition.
(ii) If $\alpha_{new} \sqsubseteq \alpha_{old}$, the result follows directly from the monotonicity of the concretization for abstract multistores. Otherwise, Let $\theta \in Cc(\alpha_{new})$. By definition of Cc for abstract multistores,

$$\exists \beta_{new} \in \alpha_{new} : \theta \in Cc(\beta_{new}).$$

If $\texttt{shape}(\beta_{new}) \notin \texttt{shape}(\alpha_{old})$, the result follows immediately. Otherwise, there exists a $\beta_{old} \in \alpha_{old}$ such that $\texttt{shape}(\beta_{new}) = \texttt{shape}(\beta_{old})$ and by Lemma 8 (ii)

$$\theta \in Cc(\beta_{new} \nabla \beta_{old}).$$

As $\beta_{new} \nabla \beta_{old} \in \alpha_{new} \nabla \alpha_{old}$, the result follows. This completes the proof for α_{new}. The proof for α_{old} is very similar.
(iii) This is because for each $\beta_{old} \in \alpha_{old}$, there is a corresponding abstract store in $\alpha_{new} \nabla \alpha_{old}$ having the same shape as β_{old}.
(iv) If $|\alpha_{old}| = |\alpha_{new} \nabla \alpha_{old}|$, then α_{new} introduces no new shapes and therefore

$$\alpha_{new} \nabla \alpha_{old} = \{\beta_{new} \nabla \beta_{old} \mid \beta_{new} \in \alpha_{new},\ \beta_{old} \in \alpha_{old},\ \texttt{shape}(\beta_{new}) \equiv \texttt{shape}(\beta_{old})\}.$$

As there is a one to one correspondence between the stores $\beta_{old} \in \alpha_{old}$ and stores $\beta_{new} \nabla \beta_{old} \in \alpha_{new} \nabla \alpha_{old}$, we have by Lemma 8 (iii) that

$$\texttt{n_infty_zero}(\alpha_{old}) \leq \texttt{n_infty_zero}(\alpha_{new} \nabla \alpha_{old}).$$

Also

$$\alpha_{new} \not\sqsubseteq \alpha_{old} \Rightarrow \exists \beta_{old} \in \alpha_{old}, \beta_{new} \in \alpha_{new} : \texttt{shape}(\beta_{old}) = \texttt{shape}(\beta_{new}) \wedge \beta_{new} \not\sqsubseteq \beta_{old}.$$

The desired result then follows by applying Lemma 8 (iv) in conjunction with part (iii). □

Theorem 13 Operation ∇ is a widening operator.

Proof: By Lemma 9 (ii),

$$\theta \in Cc(\alpha_{old}) \vee \theta \in Cc(\alpha_{new}) \Rightarrow \theta \in Cc(\alpha_{new} \nabla \alpha_{old}).$$

Let $\alpha'_0, \ldots, \alpha'_i, \ldots$ be a sequence of abstract multistores and $\alpha_0, \ldots, \alpha_i, \ldots$ a sequence of normalized abstract multistores defined as

$$\begin{array}{lcl} \alpha_0 & = & \texttt{normal}(\alpha'_0) \\ \alpha_{i+1} & = & \texttt{normal}(\alpha'_i) \ \nabla \ \alpha_i \quad (i \geq 0) \end{array}$$

We show that $\alpha_0, \ldots, \alpha_i, \ldots$ is stationary. Denote α_i by α_{old} and $\texttt{normal}(\alpha'_i)$ by α_{new}. If $\texttt{normal}(\alpha'_i) \sqsubseteq \alpha_i$, then by definition of ∇, $\alpha_{i+1} = \alpha_i$. Otherwise, since $\texttt{normal}(\alpha'_i) \not\sqsubseteq \alpha_i$, by Lemma 9 (iii) and (iv),

$$\begin{array}{l} (|\alpha_{old}| < |\alpha_{new} \nabla \alpha_{old}|) \vee \\ ((|\alpha_{old}| = |\alpha_{new} \nabla \alpha_{old}|) \ \wedge \ (\texttt{n_infty_zero}(\alpha_{old}) < \texttt{n_infty_zero}(\alpha_{new} \nabla \alpha_{old}))) \end{array}$$

This gives $\texttt{measure}(\alpha_i) < \texttt{measure}(\alpha_{i+1})$. As the set $\texttt{Measure}_n$ is finite and totally ordered for all n, there cannot be an infinite sequence in this case. □

Contents

1 Introduction **2**

2 Why Reordering is Important? **3**

3 The Concrete Domain **5**

4 The Abstract Domain `LSign` **8**
4.1 The Domain 8
4.2 Abstract Operations 13
4.2.1 Ordering of Constraint Stores 13
4.2.2 Addition of a Constraint to a Constraint Store 15
4.2.3 Upper Bound of Constraint Stores 17
4.2.4 Projection of a Constraint Store 18
4.3 Satisfiability 23
4.3.1 Satisfiability of Constraint Stores 23
4.3.2 Unsatisfiability of Constraint Stores 25
4.3.3 Conditional Satisfiability of Constraint Stores 26
4.3.4 Summary 31
4.4 The Power Domain $2^{\texttt{LSign}}$ 31
4.5 Complete Example 33
4.6 Discussion 36

5 The Abstract Domain `LInt` **36**
5.1 Abstract Objects and Concretization 37
5.2 Operations and Applications 39
5.3 The Power Domain $2^{\texttt{LInt}}$ and Widening 39
5.3.1 Informal Presentation 40
5.3.2 Formal Presentation 42
5.4 Complete Example 46
5.5 Discussion 50

6 Preliminary Experimental Results **50**
6.1 The Benchmarks 50
6.2 Execution Time 51
6.3 Compilation Time 53

7 Conclusion **53**

Bibliography **55**

Appendix: Proofs of the Results **60**