

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

An Optimizing Compiler for $\text{CLP}(\mathcal{R}_{Lin})$

by

Viswanath Ramachandran

B. Tech., Indian Institute of Technology, Bombay, 1991

Sc. M., Brown University, 1993

Thesis

Submitted in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy in the Department of Computer Science
at Brown University.

May 1998

UMI Number: 9830520

UMI Microform 9830520

Copyright 1998, by UMI Company. All rights reserved.

**This microform edition is protected against unauthorized
copying under Title 17, United States Code.**

UMI

**300 North Zeeb Road
Ann Arbor, MI 48103**

Copyright
by
Viswanath Ramachandran
1998

This dissertation by Viswanath Ramachandran
is accepted in its present form
by the Department of Computer Science
as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

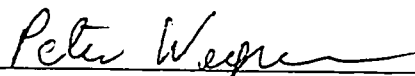
Date 9/23/97



Pascal Van Hentenryck, Director

Recommended to the Graduate Council

Date 23/9/97



Peter Wegner, Reader

Date 23/9/97



Stan Zdonik, Reader

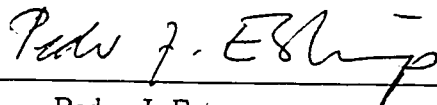
Date 23/9/97



Baudouin Le Charlier, Reader

Approved by the Graduate Council

Date Sept. 24, 1997



Peder J. Estrup
Dean of the Graduate School and Research

Vita

Viswanath Ramachandran was born on August 26th, 1970 in the Mahim suburb of Bombay (now Mumbai), India. He is the the third child of Mrs. Thangam Ramachandran and Dr. Koduvayur Viswanathier Ramachandran. He did his primary schooling (classes 1 to 7) at the University Primary School, Legon in Accra, Ghana. His secondary schooling (classes 7 to 10) was done at the Victoria High School in Mahim. His higher secondary education (classes 11 and 12) was completed at the Guru Nanak Khalsa College in Matunga, Bombay. He completed the Bachelor of Technology course in Computer Science and Engineering from the Indian Institute of Technology, Bombay in 1991 after which he joined Brown University. Viswanath received the Master of Science degree in Computer Science from Brown in 1993.

Acknowledgements

I would like to thank my advisor Pascal Van Hentenryck for his invaluable guidance and contributions to this research. Most of the results in this thesis are joint work with him. I could not have achieved this landmark without his constant comments, encouragement and constructive criticism. His time and efforts have helped to improve me as a researcher.

Thanks to Peter Wegner, Stan Zdonik and Baudouin Le Charlier for being on my thesis committee and reading this thesis. Special thanks to Baudouin Le Charlier for his comments and contributions to this research. Many of the abstract interpretation tools used in this thesis were developed by Baudouin Le Charlier and others. Agostino A. Cortesi contributed to the development and ideas behind the LInt domain, my thanks to him.

I would like to mention the late Paris C. Kanellakis who was a positive influence on me during my stay at Brown. Thanks to the faculty, staff and tstaff in the department.

My family has always been a great source of support and it is they who are responsible for who I am today. My parents have always encouraged me in my endeavours, and along with my sisters Anandi and Chitra, have been a constant source of love and affection.

I would like to thank a number of students at Brown for their support through my graduate school days. My housemates Ram Upadrasta, Santosh Kumar and Sairam Sundaram were always there for me, at the end of a hard day (or night!) in the department. My seniors Misty Nodine, Ravi Ramamurthy, Sridhar Ramaswamy, Ted Camus and Ashim Garg helped me in various ways. Swarup Acharya and Narayanan Subramanian know that the innumerable coffee breaks and long lunches we enjoyed together on Thayer Street helped me maintain my sanity in graduate school! Their concern for me will always be appreciated. Special thanks to Dimitris Michailidis and Swarup for helping me in several ways in the department.

I would also like to thank my undergraduate classmates Yezdi Lashkari, Sanjay Malpani and Vikram Dhaneshwar for making me a better Computer Scientist. My friend, Nosa Omoigui gave me a better perspective on how my graduate studies related to a career in the software industry. Gopal Kamath, Sameer Kasargod and Vikram Kansra, my buddies from Mahim were always available for long distance consultations, as evidenced by my phone bills!

Numerous teachers at University Primary School, Legon and Victoria High School, Mahim gave me a great school education. I thank them, and those institutions, for helping to mould me.

I dedicate this thesis to the millions of children who will never know the joys of a primary education, and hope that they may one day enjoy the opportunities that we take for granted.

Contents

Vita	iii
Acknowledgements	iv
List of Tables	ix
List of Figures	x
1 Introduction	1
1.1 Overview of Constraint Logic Programming	3
1.2 Motivating Examples	9
1.3 The $\text{CLP}(\mathcal{R}_{Lin})$ Runtime System	11
1.3.1 The Engine	11
1.3.2 The Interface	12
1.3.3 The Constraint Solver	13
1.4 The Challenge of CLP Optimization	13
1.5 Overview of the Compiler Optimizations	15
1.6 Program Analysis	18
1.6.1 The Abstract Interpretation Framework	18
1.6.2 The Domains LSign and LInt	21
1.6.3 The Domain Prop	24
1.7 Program Transformations	24
1.7.1 Reordering	25
1.7.2 Constraint Removal	27
1.7.3 Refinement Phase	29
1.8 Summary of Contributions	29
1.9 Organization of the Thesis	31

2	Abstract Interpretation	33
2.1	Introduction	33
2.2	Concrete Semantics	37
2.2.1	Normalized Programs	38
2.2.2	Concrete Objects	38
2.2.3	Concrete Operations	39
2.2.4	Concrete Transformation	41
2.3	Abstract Semantics	44
2.3.1	Abstract Operations	45
2.3.2	Abstract Transformation	47
2.3.3	Fixpoint	49
3	Abstract Domain LSign	51
3.1	Concrete Objects	51
3.2	Abstract Objects and Concretization	52
3.3	Abstract Operations	58
3.3.1	Ordering	59
3.3.2	Addition	62
3.3.3	Upper Bound	64
3.3.4	Projection	65
3.4	Applications	72
3.4.1	Satisfiability of Constraint Stores	72
3.4.2	Unsatisfiability of Constraint Stores	73
3.4.3	Conditional Satisfiability of Constraint Stores	75
3.4.4	Redundancy of Constraints	82
3.4.5	Freeness of Variables	84
3.5	The Power Domain 2^{LSign}	85
3.6	Relation to Abstract Interpretation Framework	87
3.7	Complete Example	88
3.8	Discussion	92
4	Abstract Domain LInt	95
4.1	Abstract Objects and Concretization	95
4.2	Operations and Applications	98
4.3	The Power Domain 2^{LInt} and Widening	98
4.3.1	Informal Presentation	99

4.3.2	Formal Presentation	102
4.4	Complete Example	106
4.5	Discussion	111
5	Abstract Domain Prop	112
5.1	Abstract Objects and Concretization	112
5.2	Abstract Operations	113
5.3	Application	115
5.4	Complete Example	115
6	Program Transformations	118
6.1	Reordering Optimization	118
6.1.1	Modified Syntax	119
6.1.2	Concepts from Operational Semantics	120
6.1.3	Admissible Reorderings	123
6.1.4	Failure-Free Reorderings	126
6.1.5	Abstract Test for Reordering	130
6.1.6	Complete Example	132
6.2	Removal Optimization	133
6.3	Refinement Optimization	134
6.4	Integration	135
7	Experimental Results	138
7.1	The Benchmarks	138
7.2	Execution Time	140
7.3	Optimization Time	142
8	Related Work	144
8.1	Optimization of CLP(\mathcal{R})	144
8.2	Abstract Interpretation of CLP	146
8.3	Optimization of Prolog	146
8.4	Optimization of Imperative Languages	146
9	Conclusion	148
9.1	Contributions	148
9.2	Open Issues	149
A	Proofs of Results	152

A.1	Ordering	152
A.2	Addition	157
A.3	Upper Bound	158
A.4	Projection	159
A.5	Satisfiability	163
A.6	Unsatisfiability	164
A.7	Conditional Satisfiability	165
A.8	Redundancy	168
A.9	Freeness	170
A.10	Widening	170
A.11	Reordering	176

List of Tables

1.1	Experimental Results: Optimizing Compiler for $\text{CLP}(\mathcal{R}_{Lin})$	32
7.1	Test Programs: Description of Usage in Various Modes.	139
7.2	Comparison of Running Times: Optimized vs. Unoptimized.	140
7.3	Optimization Times: LSign.	142
7.4	Comparison of Optimization Times: LInt vs. LSign.	143

List of Figures

1.1	Outline of the Syntax of $\text{CLP}(\mathcal{R}_{Lin})$.	3
1.2	A Simple $\text{CLP}(\mathcal{R}_{Lin})$ Program.	4
1.3	Sample Computation Tree	8
1.4	Mortgage Program.	9
1.5	Periodic Sequence Program.	10
1.6	The $\text{CLP}(\mathcal{R}_{Lin})$ Runtime System.	12
1.7	The $\text{CLP}(\mathcal{R}_{Lin})$ Constraint Solver.	14
1.8	Organization of the Compiler.	19
2.1	Simplified Concrete Semantics	42
2.2	Concrete Semantics	43
2.3	Simplified Abstract Semantics	48
2.4	Abstract Semantics	49
3.1	Concrete Projection Algorithms	66
3.2	Abstract Projection Algorithms	67
3.3	Satisfiability Algorithms	73
3.4	Unsatisfiability Algorithms	74
3.5	Reduction Algorithms	80
3.6	Redundancy Algorithms	83
3.7	Freeness Algorithms	84
3.8	Abstract Operations and Applications for 2^{LSign}	86
4.1	Modified <code>Asplit_top</code> Algorithm for <code>LInt</code>	98
6.1	Outline of the Modified Syntax of $\text{CLP}(\mathcal{R}_{Lin})$.	120
6.2	Organization of the Compiler and Optimizer	137

Chapter 1

Introduction

Constraint logic programming (CLP) [21] is a generalization of logic programming where unification is replaced by constraint solving over a suitable domain, 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., [22, 47, 8]), integers (e.g., [46]), Booleans (e.g., [6, 8]), and nonlinear real constraints (e.g., [42]). 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). The development time of such programs may be reduced significantly when compared to imperative languages, though perhaps at the expense of some run-time efficiency. The run-time efficiency is reduced because constraint solving is an expensive operation in most constraint systems.

In this thesis, efficient execution of a CLP program corresponds to mimicking the corresponding imperative implementation, wherever possible. This means that programmers need pay the overhead of using a CLP language only when their program actually needs constraint solving. When the constraint program (solving a set of constraints simultaneously) can be reimplemented as an imperative program (sequence of tests and assignments), it is our aim to perform this reimplementatation automatically. This preserves the productivity benefits of using CLP languages while not degrading runtime performance.

The thesis fits in the broad area of program optimization for special purpose languages, of which CLP is a specific instance (higher order concurrent programming and database languages being others). There are several similarities between such languages, one of which is that they are all highly declarative, making them amenable to optimization. Moreover these languages have a high potential for optimization since programmers typically specify what is to be computed and cannot express how the computation is to be performed efficiently. While this improves programmer productivity, it negatively impacts program performance, suggesting that automatic optimization of these languages is an area with practical application. Even though the thesis addresses the optimization of $\text{CLP}(\mathcal{R}_{Lin})$, the principles and results in the thesis can be used as a foundation to design optimizations for other CLP languages and other types of special purpose languages.

The techniques used in the thesis are sophisticated static analyses that collect global information about the program. The optimizations are structured as source to source transformations, and the most important optimization performed is reordering the source the program. The analyses are used to guarantee that the asymptotic complexity of programs is not changed by the reordering.

The main theoretical contributions of the thesis are the design of static analyses for collecting “interesting” information about $\text{CLP}(\mathcal{R}_{Lin})$ programs, and the statement of an admissibility criterion and test for reordering $\text{CLP}(\mathcal{R}_{Lin})$ programs. These theoretical results enable us to design and implement an optimizing compiler for $\text{CLP}(\mathcal{R}_{Lin})$, which performs provably correct optimizations and which shows that substantial performance improvements may be obtained by performing optimizations.

The rest of this chapter is organized as follows. Section 1.1 gives an overview of constraint logic programming and the language $\text{CLP}(\mathcal{R}_{Lin})$ in particular. Section 1.2 gives two examples that illustrate the basic principles underlying the use of CLP languages. The $\text{CLP}(\mathcal{R}_{Lin})$ runtime system is presented in Section 1.3. Section 1.4 motivates the optimization of CLP programs, while Section 1.5 presents the proposed optimizations for $\text{CLP}(\mathcal{R}_{Lin})$ by means of a complete worked example. Section 1.6 gives an informal review of the components used to achieve the program analysis necessary to automate the optimizations. Section 1.7 contains an informal review of how the compiler performs the optimizations automatically, given the results of program analysis. Section 1.8 summarizes the contributions of this thesis. The road map for the rest of the thesis is laid out in Section 1.9.


```

Program ::= Clauses
Clauses ::=  $\epsilon$  | Clause Clauses
Clause  ::= Head :- Body.
Head    ::= Atom
Goal    ::= Atom
Body    ::= true | OneBody, Body
OneBody ::= Goal | Constraint

```

Figure 1.1: Outline of the Syntax of $\text{CLP}(\mathcal{R}_{Lin})$.

1.1 Overview of Constraint Logic Programming

In this section, we give an overview of constraint logic programming, and of $\text{CLP}(\mathcal{R}_{Lin})$ in particular. The presentation is kept as informal as possible. Greater details can be found in [21, 22].

Syntax $\text{CLP}(\mathcal{R}_{Lin})$ programs have essentially the same syntax as Prolog, the main difference being that $\text{CLP}(\mathcal{R}_{Lin})$ programs also allow linear constraints over real numbers to appear in the bodies of clauses. Figure 1.1 shows an outline of the syntax of a $\text{CLP}(\mathcal{R}_{Lin})$ program.

A $\text{CLP}(\mathcal{R}_{Lin})$ 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 $p(t_1, \dots, t_n)$ where t_1, \dots, t_n are terms. A term is a variable (e.g. X) or a function symbol of arity n applied to n terms (e.g. $f(X, g(Y))$). 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 , terms by letters t, s , atoms by letters H, B and goals by the letter G , all possibly subscripted or superscripted. The constraints of $\text{CLP}(\mathcal{R}_{Lin})$ 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 $\delta \in \{>, \geq, =, \neq, \leq, <\}$.

$\text{CLP}(\mathcal{R}_{Lin})$ is similar to the numerical part of Prolog III [8] and is closely related to $\text{CLP}(\mathcal{R})$ [22]. To illustrate the semantics of $\text{CLP}(\mathcal{R}_{Lin})$, we use the simple program depicted in Figure 1.2.

$\begin{aligned} p(X,Y) &:- \\ &X \geq Z + 3, \\ &Y \leq Z, \\ &q(X,Y,Z). \end{aligned}$	$\begin{aligned} q(X,Y,Z) &:- \\ &r(X,Y). \\ q(X,Y,Z) &:- \\ &Z \geq Y + 2. \end{aligned}$	$\begin{aligned} r(X,Y) &:- \\ &X \leq Y + 2. \end{aligned}$
--	--	--

Figure 1.2: A Simple CLP(\mathcal{R}_{Lin}) Program.

Declarative Semantics CLP programs can be read both declaratively and operationally. Consider a clause $\text{Head} :- \text{Body}$. such that the elements of Body (goals and constraints) are b_1, \dots, b_n . Read declaratively, this clause is an implication

“ H is true if b_1 is true & ... & b_n is true”

where all the variables of the clause are universally quantified. For example, the declarative reading of the clause for p in the above program would be

“ $p(X,Y)$ is true if $X \geq Z + 3$ is true & $Y \leq Z$ is true & $q(X,Y,Z)$ is true.”

A query is a clause without a head (i.e. just a body). Consider a query $:- \text{Body}$. such that the elements of Body (goals and constraints) are b_1, \dots, b_n . This query has an answer if

“ b_1 is true & ... & b_n is true”

where all the variables of the query are existentially quantified. For example, the query

$:- Y \leq Z, r(X,Y).$

has an answer if there exist X, Y, Z such that

“ $Y \leq Z$ is true & $r(X,Y)$ is true.”

An answer to a query is an assignment of values to the variables that verifies the above existential formula. For example, $X = 2, Y = 3, Z = 4$ is an answer to the above query. Clearly, there are more answers to the above query, in fact there are infinitely many answers. To represent these answers finitely, a CLP query produces a satisfiable constraint store as the answer to a query. For example, the above query would produce the answer $Y \leq Z, X \leq Y + 2$.

Operational Semantics The operational semantics of the CLP scheme is a simple generalization of the semantics of logic programming, at least from a conceptual standpoint. It can be described as a goal-directed derivation procedure from the initial goal

using the program clauses. A *computation state* is best described by

1. a *goal part* (i.e., the conjunction of goals to be solved)
2. a *constraint store* (i.e., the set of constraints accumulated so far).

Initially the constraint store is empty and the goal part is the initial goal. In the following, computation states are denoted by pairs $\langle G \sqcap \theta \rangle$, where G is the goal part and θ the constraint store. ϵ denotes an empty goal part or constraint store. An example of a computation state occurring in the above program is

$$\langle q(X, Y, Z) \sqcap X \geq Z+3 \ \& \ Y \leq Z \rangle.$$

A *computation step* (i.e., the transition from one computation state to another) can be of two types depending upon the selection of an atom or of a constraint in the goal part. In the first case, a computation step amounts to

1. selecting an atom in the goal part;
2. finding a clause that can be used to resolve the atom; this clause must have the same predicate symbol as the atom, and the equality constraints between the goal and head arguments must be consistent with the constraint store;
3. defining the new computation state as the old one, where the selected atom has been replaced by the body of the clause and the equality constraints have been added to the constraint store.

In the second case, a computation step amounts to

1. selecting a constraint in the goal part which is consistent with the constraint store;
2. defining the new computation state as the old one where the selected constraint has been removed from the goal part and added to the constraint store.

For instance, given a computation state

$$\langle q(X, Y, Z) \sqcap X \geq Z+3 \ \& \ Y \leq Z \rangle$$

a computation step can be performed using clause 2 of q to obtain a new computation state

$$\langle Z \geq Y+2 \sqcap X \geq Z+3 \ \& \ Y \leq Z \rangle.$$

Another computation step leads to the configuration

$$\langle \epsilon \sqcap X \geq Z+3 \ \& \ Y \leq Z \ \& \ Z \geq Y+2 \rangle,$$

since the resulting constraint store is satisfiable.

As should be clear, the basic operation of the language amounts to deciding the satisfiability of a conjunction of constraints. Note also that each computation state has a satisfiable constraint store. This property is exploited inside CLP languages to avoid solving the satisfiability problem from scratch at each step. Instead, CLP languages keep a solved form of the constraints and transform the existing solution into a solution including the new constraints. Hence the constraint solver is made incremental.

A computation state is *terminal* if the goal part is empty or no clause can be applied to the selected atom to produce a new computation state or if the selected constraint cannot be satisfied with the constraint store. A *computation* is simply a sequence of computation steps that either ends in a terminal computation state or diverges. A finite computation is *successful* if the final computation state has an empty goal, and *fails* otherwise.

To illustrate computations in a CLP language, consider our simple program again. The program has only one successful computation, namely

$$\begin{array}{ll}
 \langle p(X,Y,Z) \sqcap \epsilon \rangle & \\
 \downarrow & \text{(selecting the first constraint)} \\
 \dots & \\
 \downarrow & \text{(selecting the last constraint)} \\
 \langle q(X,Y,Z) \sqcap X \geq Z+3 \ \& \ Y \leq Z \rangle & \\
 \downarrow & \text{(using clause 2 of } q \text{)} \\
 \langle Z \geq Y+2 \sqcap X \geq Z+3 \ \& \ Y \leq Z \rangle & \\
 \downarrow & \text{(selecting the constraint)} \\
 \langle \epsilon \sqcap X \geq Z+3 \ \& \ Y \leq Z \ \& \ Z \geq Y+2 \rangle &
 \end{array}$$

The program also has one failed computation:

$$\begin{array}{ll}
 \langle p(X,Y,Z) \sqcap \epsilon \rangle & \\
 \downarrow & \text{(selecting the first constraint)} \\
 \dots & \\
 \downarrow & \text{(selecting the last constraint)} \\
 \langle q(X,Y,Z) \sqcap X \geq Z+3 \ \& \ Y \leq Z \rangle &
 \end{array}$$

↓	(using clause 1 of <i>q</i>)
$\langle r(X, Y) \square X \geq Z+3 \ \& \ Y \leq Z \rangle$	
↓	(using clause 1 of <i>r</i>)
$\langle X \leq Y+2 \square X \geq Z+3 \ \& \ Y \leq Z \rangle$	

The last computation state is terminal since the conjunction of constraints

$$X \geq Z+3 \ \& \ Y \leq Z \ \& \ X \leq Y+2$$

is not satisfiable.

Note that the results of the computation are the constraint stores of the successful computations (possibly projected on the query variables). For example, an answer to the query

$:- p(X, Y).$

is the projection of $X \geq Z+3 \ \& \ Y \leq Z \ \& \ Z \geq Y+2$ on the variables *X* and *Y*, giving the constraint store $X \geq Y+5$.

Also, nothing has been said so far on the strategy used to explore the space of computations. Most CLP languages use a computation model similar to Prolog: atoms are selected from left to right in the clauses; clauses are tried in textual order; and the search space is explored in a depth-first manner with chronological backtracking in case of failures. For instance, on the simple program, a CLP language typically uses clause (1) for *p*, then clause (1) for *q*, and finally encounters a failure when trying to solve *r*. Execution then backtracks to clause (2) of *q*, giving a successful computation.

In order to summarize the information given in this section informally, it is useful to refer to the *computation tree* for the query $:- p(X, Y)$ which is given in Figure 1.3. The nodes of the computation tree represent various computation states (with the root node representing the query), while the edges of the tree represent computation steps. Leaf nodes of the tree represent terminal computation states, while paths beginning at the root node represent computations. The strategy used to explore the space of computations is reflected in the left to right ordering of the children of a node. The tree is also annotated to indicate the various clauses used and the results of the computations.

Useful Features of CLP Languages As mentioned previously, CLP languages are useful for developing solutions to a variety of combinatorial search problems. In particular, *multi-directionality* of programs and *nondeterminism* combined with *constraint*

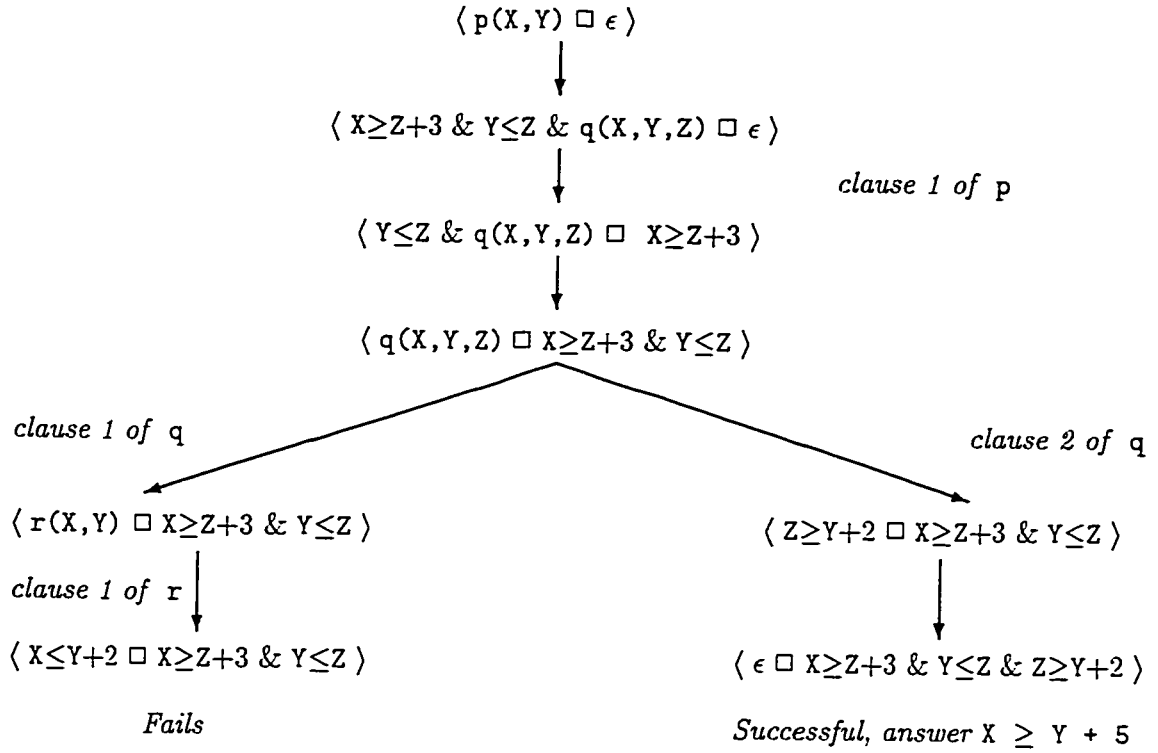


Figure 1.3: Sample Computation Tree

solving are features that enable rapid development of solutions to many problems. Multi-directionality is a feature borrowed from logic programming. Informally, it means that an argument to a predicate may be either input or output depending upon the context of its use. As a result, the same program can be used in a variety of ways without any modifications. The mortgage example in the next section exemplifies the advantages of multi-directionality. Nondeterminism is also borrowed from logic programming and is manifested by multiple clauses in the definition of predicates. As described in the previous section, it is simulated by systematically searching the space of computations. For the programmer, nondeterminism removes the burden of programming a search for the solution. Incremental constraint solving is the basic operation of CLP, and is a generalization of unification in logic programming, which can be seen as a special case of constraint solving. For programmers, having constraint solving as a basic operation of the language removes the burden of programming constraint-solving techniques, and enables them to focus on modeling the problem with appropriate constraints. The combination of nondeterminism and constraint solving is a useful programming tool and is well illustrated by the periodic sequence example in the next section.

```

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

```

Figure 1.4: Mortgage Program.

1.2 Motivating Examples

In order to illustrate the basic principles underlying the use of CLP languages, we shall present a couple of more complicated CLP(\mathcal{R}_{Lin}) programs.

Mortgage The first program is the mortgage example [22], which relates various parameters in a mortgage computation. Figure 1.4 presents a CLP program for the mortgage example. 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 program is its multi-directionality. For example, the query

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

attempts to compute 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. The predicate `mg` can be used in a multitude of ways because of the general constraint solver embedded in the language.

```

invalidate :-
    [X1,X2] ≠ [X10,X11], sequence([X11,X10,X9,X8,X7,X6,X5,X4,X3,X2,X1]).

sequence([X2,X1]).
sequence([XiPlus2,XiPlus1,Xi|Xs]) :-
    abs(XiPlus1,XiPlus1a), XiPlus2 = XiPlus1a - Xi,
    sequence([XiPlus1,Xi|Xs]).

abs(X,X) :- X ≥ 0.
abs(X,-X) :- X < 0.

```

Figure 1.5: Periodic Sequence Program.

Periodic Sequence The second program [8] is a mathematical problem which highlights many of the functionalities of $\text{CLP}(\mathcal{R}_{Lin})$. The problem involves an infinite sequence of numbers x_1, x_2, \dots defined by

$$x_{i+2} = |x_{i+1}| - x_i \quad (i \geq 1)$$

where x_1 and x_2 are arbitrary real numbers. The problem consists of showing that the sequence is always periodic and has a period of 9 or, in other words, that the sequences x_1, x_2, \dots and x_{10}, x_{11}, \dots are always identical. The problem can be solved indirectly by using the following idea: since the two sequences are completely determined as soon as their first two elements are fixed, it is sufficient to show that any sequence $x_1, x_2, \dots, x_{10}, x_{11}$ implies $x_1 = x_{10}$ and $x_2 = x_{11}$. The program then consists in searching for a solution such that $\langle x_1, x_2 \rangle \neq \langle x_{10}, x_{11} \rangle$. The absence of solutions proves the conjecture. The complete program is shown in Figure 1.5. The predicate `sequence` defines the sequence in reverse order. It is defined recursively and uses the predicate `abs` to compute the absolute value. `abs` is nondeterministic (since the values are not known) and first enforces the constraint that its argument is positive. On backtracking, `abs` enforces the constraint that its argument is strictly negative. Note that the program never assigns any value to the variables. Yet the program fails in all branches of the search tree thanks to the complete constraint solver and its ability to deal exactly with strict inequalities.

1.3 The $\text{CLP}(\mathcal{R}_{Lin})$ Runtime System

The implementation of $\text{CLP}(\mathcal{R}_{Lin})$ is organized around three modules, the engine, the interface, and the constraint system, with a top-down communication pattern. The engine is responsible for the control part of the system (e.g. clause and goal selection) and is a traditional WAM based Prolog system. The constraint system is responsible for constraint solving. The interface is responsible for communication between the engine and constraint solver. It converts the constraints encountered by the engine during execution from the engine representation (i.e. using tagged values to represent variables) into their solver representation (i.e. using natural numbers to identify variables), and sends them to the constraint solver. It receives a Boolean return value from the constraint solver that indicates the success or failure of the constraint solving, and communicates the return value back to the engine. In addition, the interface solves the simplest constraints by itself, without communicating with the solver, and returns a Boolean value to the engine. The interface also receives choice point instructions from the engine (try, retry, and trust) which it communicates to the constraint solver in order that the constraint solver can maintain its own data structures and choice point stacks for supporting backtracking. Design decisions similar in spirit were adopted in $\text{CLP}(\mathcal{R})$ [22]. More details about the $\text{CLP}(\mathcal{R}_{Lin})$ runtime system are available in [49]. The overall organization of the runtime system into various modules is illustrated in Figure 1.6.

1.3.1 The Engine

In the following, we assume that linear constraints $t_1 \delta t_2$ where $\delta \in \{<, \leq, =, \neq, \geq, >\}$ are rewritten into the form $0 \delta t$ by an earlier phase of the compiler. In the engine, the code for executing a linear constraint $0 \delta t$ is a sequence of instructions that gives the structure of the term t (i.e. the coefficients of the variables and the constant term), followed by an instruction that gives the type of constraint δ . For example, the sequence of instructions generated for the constraint $4X_1 + 5X_2 = 3X_1 + 2X_3 + 4$ in the source program is

```
ADD_VARIABLE -1 X1
ADD_VARIABLE -5 X2
ADD_VARIABLE 2 X3
ADD_CONSTANT 4
CONSTRAINT_EQUAL
```

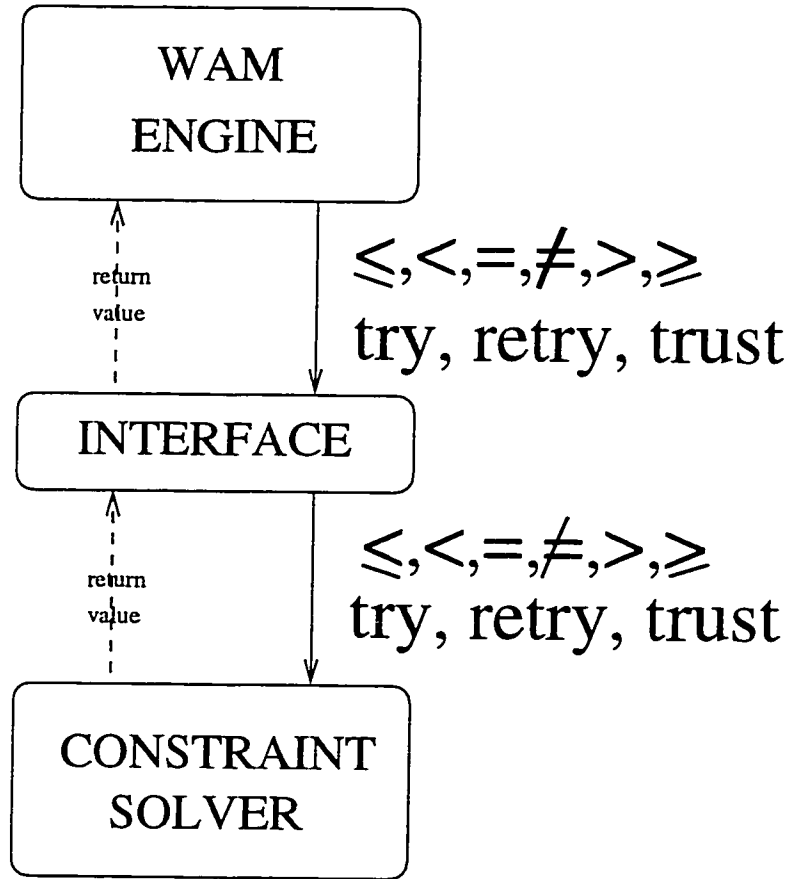


Figure 1.6: The CLP(\mathcal{R}_{Lin}) Runtime System.

The variables in the engine are tagged to indicate whether they occur in the constraint solver or not.

1.3.2 The Interface

The interface receives the linear term constructed by the engine (using `ADD_VARIABLE` and `ADD_CONSTANT`), converts it into a representation suitable for the solver and sends it to the solver along with the type of constraint (eg. `CONSTRAINT_EQUAL`). The interface first simplifies some of the constraints using the variable binding information of the engine. For example, when executing `ADD_VARIABLE -5 X2`, if the tag of X_2 indicates that it is an integer or rational, the value of X_2 (eg. 1) is used by the interface to simplify the term to $-1X_1 + 2X_2 - 1$. In addition, if after receiving the linear term from the engine, it is found to consist entirely of constants and ground variables, a simple test can be

performed in the interface itself to see if the constraint is satisfiable. For example in the above linear term, if X_1 , X_2 and X_3 are bound to 2, 1 and 3 respectively, then the interface only needs to check if $(-1 \times 2 - 5 \times 1 + 2 \times 3 + 4 = 0)$. In the case of equations, they can also be simplified to assignments if one of the variables is unconstrained (does not occur in the constraint solver) and all the other variables are ground. For example, if X_2 and X_3 have ground values and X_1 is a free variable (i.e. not constrained), then the equation above becomes an assignment $X_1 := -5X_2 + 2X_3 + 4$. i.e. the linear expression on the R.H.S. can be evaluated and assigned to X_1 .

1.3.3 The Constraint Solver

The constraint solver and solved form are described in detail in [49]. The constraint solver receives constraints from the interface and updates its accumulated constraint store. Each time a constraint is added to the solver, it returns a Boolean value indicating whether the constraint was satisfiable in conjunction with the accumulated store. The constraint solver is organized into two modules, the Gauss (G) subsystem, and the Simplex (S) subsystem. Briefly, the Gauss subsystem is used for solving equations, while the Simplex subsystem is used for solving inequalities. The solved form in the Gauss subsystem consists of isolating one variable per equation (called the *basic* variable) and eliminating it from the rest of the store. The solved form in the Simplex subsystem is similar, but it also imposes a lexicographic requirement [47] on the equations. In both subsystems disequations are fully dereferenced (i.e the basic variables are eliminated from them) and required to be different from $0 \neq 0$. It can be proved that a system of constraints over \mathcal{R}_{Lin} is satisfiable if and only if it can be expressed in a solved form. Therefore, the constraint solving algorithms for $CLP(\mathcal{R}_{Lin})$ consist of maintaining the accumulated store in a solved form incrementally. The organization of the constraint solver into two modules is illustrated in Figure 1.7. More details about the constraint solver are available in [49].

1.4 The Challenge of CLP Optimization

While CLP languages greatly reduce the development time of several classes of programs (especially programs for combinatorial search problems), this reduction in development time may come at the expense of some run-time efficiency. This is because constraint solving is an expensive operation in general and the generic constraint solver used in a

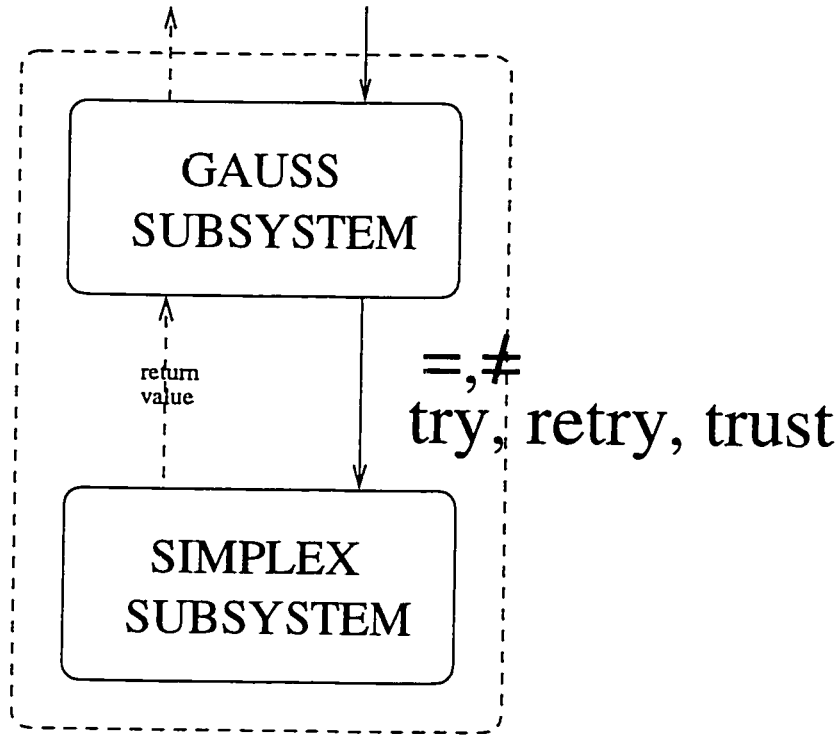


Figure 1.7: The $\text{CLP}(\mathbb{R}_{Lin})$ Constraint Solver.

CLP language may be unable to exploit the features of a specific program, unlike an imperative program with custom constraint-handling code. The main challenge of CLP optimization is to bridge the gap in efficiency between CLP and imperative language implementations. Improving the efficiency of CLP programs should encourage more widespread use of CLP in the real world.

Our approach to CLP optimization is similar to the 3R's methodology proposed by Marriott and Stuckey [36] for $\text{CLP}(\mathbb{R})$. The approach which generalizes similar methodologies for logic programming, consists of refining constraints into tests and assignments, removing redundant constraints (i.e., constraints which are implied by the constraint store), and reordering constraints to maximize refinements and removals. The optimizations are specific to the domain \mathbb{R} of real constraints but similar optimizations can be applied to other domains as well. Reordering is the most delicate optimization, since the compiler must make sure that the resulting program preserves the same search tree for the programs. This in turn guarantees to preserve the termination of the resulting program and makes sure that the “optimized” program can never be significantly slower than the unoptimized program.

The aim of this thesis is to present the design and implementation of the first optimizing compiler for $\text{CLP}(\mathcal{R}_{Lin})$ implementing refinements, reordering, and removal of redundant constraints. The optimizations are global in nature, and require information about the macro properties of the program in order to be automated successfully. The compiler (40,000 lines of C, including about 10,000 lines only for the optimizations), uses abstract interpretation [10] to collect the information necessary to perform the optimizations safely. In particular, it uses abstract interpretation algorithms derived from the generic abstract interpretation system GAIA, [29] instantiated to the following abstract domains: Prop [34, 52] to deduce information on fixed variables, and LSign [37, 43] (or LInt) to deduce unconstrained variables, redundant constraints, and satisfiable constraint stores.

1.5 Overview of the Compiler Optimizations

The high-level optimizations performed by our compiler may be viewed as source to source transformations, 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 $\text{Var} := \text{Exp}$ which, when executed, assumes that Var is an unconstrained variable (i.e., it does not occur in the accumulated constraint store) while Exp is a fixed expression (i.e., an expression whose variables are fixed to a value). In the runtime model of the language, we must have for assignments that Var does not occur in the constraint store, and all variables in the linear expression Exp must be bound to a value in the WAM engine.¹ This means that the right side expression can be evaluated without using the constraint solver and the evaluated value can be bound to the left side variable. A test is of the form $\text{Exp1} \text{ ?}\delta \text{ Exp2}$, where $\delta \in \{=, <, \leq, >, \geq, \neq\}$ and where Exp1 and Exp2 are fixed expressions. The compiler is organized in four main phases: normalizing, reordering, removal, and refinement.

To illustrate the various phases, we consider the mortgage example presented earlier. This program shall constitute the running example to illustrate various concepts throughout the thesis. The predicate $\text{mg}(\text{P}, \text{T}, \text{R}, \text{B})$ relates a mortgage's principal (P), number of installments (T), monthly repayment (R) and final balance (B) as follows:

¹Our implementation does not do this exactly. It is possible for variables that have their value fixed in the constraint solver to appear on the right side of assignments. This requires us to look up the fixed value from the constraint solver in that case. It is possible to eliminate this requirement by doing a more accurate analysis for assignments.

$\text{mg}(P, 0, R, P).$

$\text{mg}(P, T, R, B) :- T > 0, P \geq 0, \text{mg}(P * 1.01 - R, T - 1, R, B).$

The program is multi-directional and can be used in various modes. We illustrate the optimizations when mg is used with P and R fixed (i.e., they are constrained to take a value) and T and B are unconstrained. Note that the various uses of a predicate in a program can be obtained automatically by abstract interpretation² and that several versions of the program can be generated when the predicate is used in multiple ways.

The first phase of our compiler consists of normalizing the program to make constraints explicit in order to ease analysis and optimization. On our running example, this phase produces the program

$\text{mg}(P, T, R, B) :- T = 0, B = P.$

$\text{mg}(P, T, R, B) :- T > 0, P \geq 0, P_1 = P * 1.01 - R, T_1 = T - 1, \text{mg}(P_1, T_1, R, B).$

The second phase of our compiler tries to move constraints to a place in the clause where, roughly speaking, they can be specialized into tests or into assignments (a test or assignment can be implemented so that only variable bindings in the interface or WAM are utilized, rather than the constraint solver). 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.³ On our running example, our 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 input constraint store for mg and every intermediate constraint store that occurs in the execution of the program until after the recursive call to mg . Hence, $T > 0$ cannot prune the search space. Our compiler proves this automatically by the LSign analysis. Note also that, when postponed until after the recursive call, $T > 0$ may be specialized into a test, since T is fixed. Similarly, $T_1 = T - 1$ 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:

²This assumes of course that users specify the top-level input pattern which, roughly speaking, specifies which arguments are results and which are data.

³It is extremely difficult to guarantee that the “optimized” program is at least as efficient as the unoptimized program, since the constraint solver (based on the simplex algorithm) may be sensitive to the ordering of the constraints. No case of slowdowns was observed on our benchmarks however.

$\text{mg}(P,T,R,B) :- T = 0, B = P.$

$\text{mg}(P,T,R,B) :- P \geq 0, P1 = P*1.01 - R, \text{mg}(P1,T1,R,B), T1 = T - 1, T > 0.$

It is important to note that there may be several places where a constraint can be moved. Our compiler uses simple heuristics to decide where to attempt to move a constraint.

The third phase of the compiler 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.⁴ Once again, this fact can be proven automatically through abstract interpretation using the domain LSign. The resulting program is as follows:

$\text{mg}(P,T,R,B) :- T = 0, B = P.$

$\text{mg}(P,T,R,B) :- P \geq 0, P1 = P*1.01 - R, \text{mg}(P1,T1,R,B), T1 = T - 1.$

The fourth phase of the compiler 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 $\text{Var} = \text{Exp}$ 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 . 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.

$\text{mg}(P,T,R,B) :- T := 0, B := P.$

$\text{mg}(P,T,R,B) :- P \geq 0, P1 := P*1.01 - R, \text{mg}(P1,T1,R,B), T := T1 + 1.$

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 [12] can be used to transform our final program into a tail-recursive program. Similarly, efficient instructions can be generated for the tests and assignments [23]. These additional optimizations are orthogonal to the contents of this thesis and are not discussed here.

⁴Note that this constraint is useful for other uses of the program.

The overall organization of the compiler into various phases is illustrated in Figure 1.8.

1.6 Program Analysis

Our compiler performs a series of analyses to infer the runtime properties necessary to carry out the optimizations. It uses abstract interpretation [10], a systematic method to develop static analyses. In particular, the compiler uses algorithms based on the generic abstract interpretation system GAIA [29], instantiated to the following domains: Prop to determine fixed variables and LSign (or LInt) to determine unconstrained variables, redundant constraints and satisfiable constraint stores. This section contains an informal review of these components.

1.6.1 The Abstract Interpretation Framework

The abstract interpretation framework used by the compiler is a natural extension to CLP of the logic programming framework in [29]. It follows the traditional approach to abstract interpretation [10].

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

for every computation beginning at $\langle p(x_1, \dots, x_n) \sqcap \theta_{in} \rangle$, where $\theta_{in} \in \Theta_{in}$:

1. for every terminal computation state $\langle \epsilon \sqcap \theta_{out} \rangle$: $\theta_{out}/\{x_1, \dots, x_n\} \in \Theta_{out}$;
2. for every intermediate computation state $\langle G \sqcap \theta_{int} \rangle$: $\theta_{int}/\{x_1, \dots, x_n\} \in \Theta_{int}$.

Intuitively, Θ_{out} is the set of all possible output constraint stores (projected on the initial query variables) when p is executed with any store from Θ_{in} as the input. Θ_{int} is the set

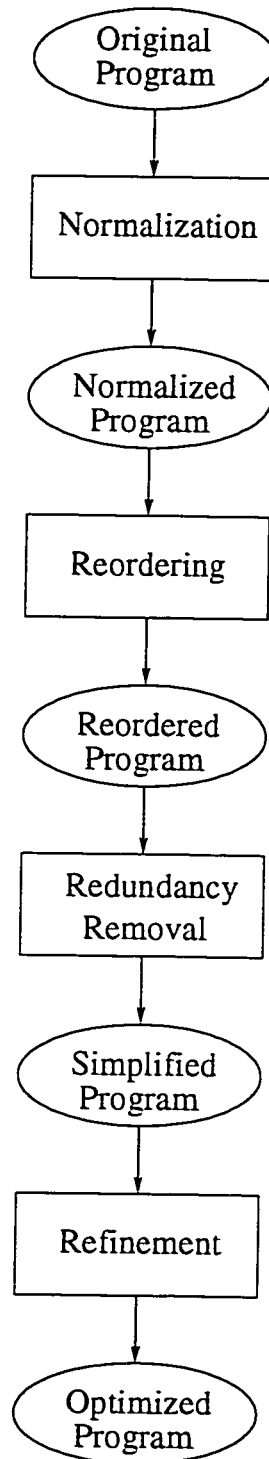


Figure 1.8: Organization of the Compiler.

of all possible intermediate constraint stores (projected on the initial query variables), i.e. accumulated constraint store for any intermediate computation state, when p is executed with any store from Θ_{in} as the input. The concrete semantics is similar to that for logic programs in [29], except that it has a component for the intermediate descriptions in addition to the output descriptions.

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

for every computation beginning at $\langle p(x_1, \dots, x_n) \sqcap \theta_{in} \rangle$, where θ_{in} is a constraint store satisfying the property expressed by β_{in} :

1. for every terminal computation state $\langle \epsilon \sqcap \theta_{out} \rangle$: $\theta_{out}/\{x_1, \dots, x_n\}$ is a constraint store that satisfies the property expressed by β_{out} ;
2. for every intermediate computation state $\langle G \sqcap \theta_{int} \rangle$: $\theta_{int}/\{x_1, \dots, x_n\}$ is a constraint store that satisfies the property expressed by β_{int} .

Intuitively, β_{out} represents all the constraint stores that can be the output, and β_{int} represents all the intermediate constraint stores. when p is executed with any store satisfying β_{in} as the input. β_{in} is called an input store of p while β_{out} is called an output store and β_{int} is called an intermediate store. In this approach, the link between the abstract and the concrete domain is given by a monotone concretization function. The abstract semantics assumes a number of operations on abstract stores, in particular addition of a constraint, projection, and upper bound. The first two operations are consistent approximations of the corresponding concrete operations. The upper bound operation is a consistent abstraction of union of sets of constraint stores.

The last step of the methodology consists of computing the least fixpoint or a post-fixpoint of the abstract semantics using an algorithm such as GAIA [29] or PLAI [40]. Both of these are top-down algorithms computing a small, but sufficient, subset of least fixpoint (or of a postfixpoint) necessary to answer a user query.

From the abstract interpretation results, our optimizing compiler uses basically three pieces of information for each predicate p in the program. First, it uses the output store β_{out} describing the constraint stores obtained by running p on an empty constraint store. This output store is called the *output description* of p . Second, it collects all the

input abstract stores β_{in} encountered when analyzing the program for the query. These abstract stores, which describe all possible constraint stores than can be encountered when p is called, are summarized into a single abstract store that we call the *input description* of p . This input description of course characterizes all input stores for p encountered at runtime. Finally, it uses the intermediate store β_{int} describing all the intermediate constraint stores that can occur during the execution of p on an empty constraint store. This store is called the *intermediate description* of p . The fact that these three descriptions suffice to produce good results comes from the nature of the abstract domains.

1.6.2 The Domains LSign and LInt

The domain LSign [37, 43] is fundamental for the reordering and redundancy phases and for detecting unconstrained variables in the refinement phase. Its two critical ideas are the replacement of coefficients by their signs and the association of multiplicity information with constraints. A sign is an element of $\{0, \oplus, \ominus, \top\}$, where \oplus denotes the (strictly) positive real numbers, \ominus denotes the (strictly) negative real numbers, 0 denotes zero, and \top denotes all real numbers. An abstract constraint is an expression of the form $s_0 \text{ op } \sum_{i=1}^n s_i x_i$ where s_i is a sign and op is a relational operator (eg., \leq). An abstract constraint denotes the set of constraints obtained by replacing the signs by coefficients in their denotations.

Example 1 The abstract constraint $\oplus = \ominus x_1 + \top x_2$ represents both the constraint $3 = -x_1 + x_2$ and $3 = -x_1 - x_2$ but not the constraint $3 = x_1 - x_2$.

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. An abstract constraint with multiplicity is the association of an abstract constraint and a multiplicity. It denotes sets of abstract stores (i.e., multiset of constraints).

Example 2 The abstract constraint with multiplicity $\langle \oplus = \ominus x_1 + \top x_2, \text{One} \rangle$ represents only multisets of size 1, e.g., $\{3 = -x_1 + x_2\}$. $\langle \oplus = \ominus x_1 + \top x_2, \text{Any} \rangle$ represents multisets of any size, e.g., \emptyset , $\{3 = -x_1 + x_2\}$ and $\{3 = -x_1 + x_2, 3 = -x_1 - x_2\}$.

An abstract store is a set of abstract constraints with multiplicities. An abstract store obviously denotes a set of constraint stores. An *LSign* description is a finite set of abstract stores. Note that it would be sufficient to use abstract stores as *LSign* descriptions but the use of their powerset as abstract domain enhances precision significantly in many examples.

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

Note that abstract operations on this domain are non-trivial and use abstract versions of Fourier and Gaussian elimination. The details are presented later in this thesis and also in [37, 43]. We now illustrate the domain *LSign* on our running example. The top-level query, where *P* and *R* are fixed, is associated with the abstract store

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

stating that *P* and *R* are fixed to a real number. The output description for program *mg* is as follows:

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

The first abstract store represents the results of the first clause. It requires the time period to be zero and it imposes the constraint linking the remaining variables. The second abstract store is the result of the second clause, when the recursive call uses the first clause once. The third abstract store is the result of the second clause when the recursive call uses the second clause at least once (and finitely often) and the first clause once. The input description for *mg*, which captures all constraint stores used as input

for mg, is as follows:

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

It contains of course the query but also many other abstract stores which characterize constraint stores occurring at runtime as input to mg. For instance, the second abstract store captures the stores encountered for the first recursive call to mg. The intermediate description for mg, which captures all the stores that can occur during a computation beginning with the initial computation state $\langle \text{mg}(P, T, R, B) \sqcap \epsilon \rangle$ is as follows:

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

The first abstract store represents the state when entering the predicate mg, i.e. with no accumulated constraints. The last three abstract stores represent the state when leaving the predicate mg, i.e. the output description. The other stores represent various possible intermediate states. The second and third stores represent the state after executing the first constraint of the first and second clause respectively. The fourth store represents the state after executing the first two constraints of the second clause. The fifth, sixth and seventh abstract stores represent intermediate stores that can occur inside various recursive calls to the predicate mg. As can be seen, even a fairly simple program can have a very complicated description of all its intermediate states.

The domain `Lint` generalizes `LSign` by abstracting coefficients by intervals instead of signs. This enables a more precise analysis of programs in many cases. This will be discussed in greater detail later in the thesis. In general, we continue the presentation only with the domain `LSign`, for simplicity.

1.6.3 The Domain Prop

The domain `Prop` [35, 52] is an effective domain to compute groundness information for Prolog. It can be extended easily to infer fixed variables in CLP (e.g. [17]). Its key idea is to represent the information through a Boolean formula. Informally speaking, a formula $x \leftrightarrow y$ means that, whenever x is fixed, y is fixed and vice-versa. A formula $x \wedge y \rightarrow z$ means that, whenever x and y are fixed, z is fixed as well. A formula x means that x is fixed. An equation $x = y$ is abstracted by a formula $x \leftrightarrow y$, and an equation $a_0 = a_1x_1$ is abstracted by a formula x_1 , while an equation $a_0 = a_1x_1 + \dots + a_nx_n$ is abstracted by a set of formulas

$$\begin{aligned} & x_1 \wedge \dots \wedge x_{n-1} \rightarrow x_n, \\ & x_1 \wedge \dots \wedge x_{n-2} \wedge x_n \rightarrow x_{n-1}, \\ & \dots \\ & x_2 \wedge \dots \wedge x_n \rightarrow x_1. \end{aligned}$$

Inequalities are abstracted by 1, i.e., they are not used to infer fixed variables. We now illustrate the domain `Prop` on our running example. The output description for `mg` is the Boolean formula

$$T \wedge (R \wedge B \rightarrow P) \wedge (P \wedge R \rightarrow B)$$

It states that `T` has a fixed value and specifies the relationship between the other variables. The input description for `mg` is the Boolean formula

$$P \wedge R$$

stating that all calls to `mg` have fixed values for `P` and `R`. The intermediate description for `mg` is just the Boolean formula 1, indicating that it cannot be deduced by `Prop` as to whether any of the variables take a fixed value for any arbitrary intermediate computation state while executing `mg` with an initially empty constraint store.

1.7 Program Transformations

In this section, we discuss informally how the compiler performs the reordering, removal, and refinement optimizations, given the availability of the program analysis information

discussed in the previous section.

1.7.1 Reordering

Reordering is the most complicated phase of the compiler. This phase performs LSign and Prop analyses on the normalized program. The Prop analysis is necessary to identify where to move constraints, while the LSign analysis is needed to determine if the move is acceptable. First of all, note that declaratively, reordering does not pose any problems, because the original and reordered programs return the same answers. Therefore, to understand when a reordering is possible, it is necessary to consider the operational semantics. Consider a clause

$$p \text{ :- } g_1, \dots, g_i, \lambda, g_{i+1}, \dots$$

and assume that the compiler is interested in moving constraint λ after goal g_{i+1} . Consider an execution of the clause, and let θ be the accumulated constraint store at the point just before the constraint λ . Intuitively, it is acceptable to move λ after g_{i+1} if we can guarantee that doing so does not increase the execution time substantially, as compared to the original program. Our approach is to guarantee that the original program and the reordered programs have the same search space. This is true if the following two criteria are met:

1. λ does not prune the search space at the program point just after g_i ; and
2. λ does not prune the search space inside the execution of g_{i+1}

More precisely, if θ is the accumulated constraint store at the program point just after g_i , it is acceptable to move λ after g_{i+1} if

1. $\theta \sqcup \{\lambda\}$ is satisfiable; and
2. for any intermediate computation state $\langle G \sqcap \theta_i \rangle$ of a computation starting at $\langle g_{i+1} \sqcap \theta \rangle$, $\theta_i \sqcup \{\lambda\}$ is satisfiable.

The second criterion subsumes the first, and so it is sufficient to verify the second criterion alone. The reordering algorithm attempts to carry out the above proof obligation using the LSign analysis. The analysis produces an LSign description that captures all the constraint stores that can occur in the computation starting at $\langle g_{i+1} \sqcap \theta \rangle$, call this

description α_i . In order to understand how the proof obligation is verified in LSign, it is instructive to first see how the corresponding property is verified in the concrete domain. If θ_i is a constraint store, checking if $\theta_i \sqcup \{\lambda\}$ is satisfiable consists of the following steps:

1. Add λ to θ_i in order to obtain θ' ;
2. Eliminate all variables from θ' to produce θ_r ;
3. Test if θ_r is always satisfiable, i.e., if it contains only trivially satisfiable constraints (eg. $0 = 0$, $c \leq 0$ where c is a non-positive number etc.)

If so, then $\theta_i \sqcup \{\lambda\}$ is satisfiable. Verifying the property in LSign is similar. The only difference is that it is advantageous to first express the LSign description α_i on the variables of the constraint λ , in order to make the information more explicit. This increases the precision of the abstract test. Also it is only necessary to consider the satisfiable stores in α_i . As a consequence, it is useful to eliminate obvious sources of inconsistencies from α_p (e.g., all constraints without variables) to increase the precision of the abstract test. The proof obligation in LSign is as follows.

0. Project α_i on the variables of λ and remove constraints without variables to obtain α_p ;
1. Add the abstraction of λ (i.e., the constraint where the coefficients have been replaced by signs) to α_p in order to obtain α' ;
2. Eliminate all variables from α' to produce α_r ;
3. Test if α_r is always satisfiable, i.e., if it contains only constraints of the form $\ominus < 0$, $\ominus \leq 0$ etc.

We now illustrate this on our running example. Consider the normalized program

```
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).
```

and assume that the compiler tries to move constraint $T > 0$ after the recursive call. This actually consists of a sequence of reorderings, the first of which attempts to move $T > 0$ after the constraint $P \geq 0$. The first proof obligation consists of checking whether $T > 0$ is always consistent with all the stores that can occur in a computation beginning

at $\langle P \geq 0 \sqcap \theta \rangle$, where θ is any computation state that can occur just before $T > 0$. These computation states can be captured by the abstract description α_i , where $\alpha_i = \alpha_{in} \cup \alpha_2$, α_{in} is the input store of *mg* depicted in Section 1.6.2, and α_2 is the abstract description obtained by adding the constraint $P \geq 0$ to α_{in} . When α_i is projected on T , and constraints without variables are removed, α_i gives α_p :

$$\{\{\}, \{\langle \ominus < \oplus T, \text{One} \rangle\}, \{\langle \ominus < \oplus T, \text{One} \rangle, \langle \ominus < \oplus T, \text{Any} \rangle\}\}.$$

Adding the abstraction of the constraint $T > 0$ produces α' :

$$\begin{aligned} &\{ \\ &\quad \{\langle 0 < \oplus T, \text{One} \rangle\}, \\ &\quad \{\langle \ominus < \oplus T, \text{One} \rangle, \langle 0 < \oplus T, \text{One} \rangle\}, \\ &\quad \{\langle \ominus < \oplus T, \text{One} \rangle, \langle \ominus < \oplus T, \text{Any} \rangle, \langle 0 < \oplus T, \text{One} \rangle\} \\ &\} \end{aligned}$$

All the stores in the concretization are obviously satisfiable, since the projection gives us the empty set of constraints (i.e. $\alpha_r = \{\}$). The next two reordering steps involve moving $T > 0$ after the constraints $P1 = P * 1.01 - R$ and $T1 = T - 1$ and similar proof obligations can be carried out. The final reordering step consists of moving $T > 0$ after the recursive call to *mg*. We demonstrate the proof obligation which verifies that $T > 0$ does not prune the search space inside the recursive call to *mg*. The store α' in the proof obligation (representing all possible satisfiable intermediate computation states that can occur in the recursive call to *mg*, projected on T) is

$$\begin{aligned} &\{ \\ &\quad \{\}, \\ &\quad \{\langle \ominus < \oplus T, \text{One} \rangle, \langle \ominus < \oplus T, \text{Any} \rangle\}, \\ &\quad \{\langle \oplus < \oplus T, \text{One} \rangle, \langle \oplus < \oplus T, \text{Any} \rangle\}, \\ &\quad \{\langle \ominus < \oplus T, \text{One} \rangle, \langle \ominus < \oplus T, \text{Any} \rangle, \langle \oplus < \oplus T, \text{One} \rangle, \langle \oplus < \oplus T, \text{Any} \rangle\}, \\ &\quad \{\langle \oplus = \oplus T, \text{One} \rangle\} \\ &\} \end{aligned}$$

which can easily be seen to be consistent with $T > 0$.

1.7.2 Constraint Removal

This phase receives the reordered program and performs the LSign analysis on the reordered program. Each constraint in each clause is then considered for constraint removal. To understand when a constraint can be removed from the program, it is useful

to review the runtime system of the language. Consider a clause

$$p :- g_1, \dots, g_i, \lambda, g_{i+1}, \dots$$

and assume that the compiler is interested in determining whether the constraint λ can be removed from the body of the clause. Consider an execution of the clause, and let θ be the accumulated constraint store at the point just before λ . Before trying to add λ to the constraint store, the constraint solver first seeks to determine if λ is redundant w.r.t. the store θ ; in that case, the constraint does not supply any new information and so it need not be added to the store at all. The following actions are taken to determine if λ is redundant w.r.t. θ .

1. Simplify λ using the equations of θ , i.e. project the non-basic variables of the store from the constraint.
2. Test if the simplified constraint is trivially satisfiable (e.g. $0 = 0$, $c < 0$ where c is a negative number, etc.)

If so, then the constraint is redundant w.r.t. the store.

The constraint removal phase attempts to mimic this at compile time, using the LSign analysis. The analysis produces an LSign description that represents all the constraint stores θ that can occur at the program point just before λ ; call this description α . As before, it is advantageous to first express α on the variables of λ so as increase the precision of the abstract test. The following actions are taken to determine if λ is redundant.

0. Project α on the variables of λ to obtain α' ;
1. Simplify the abstraction of λ w.r.t. α' by using the equations of α' ;
2. Test if the simplified abstraction of λ is trivially satisfiable (e.g. $0 = 0$, $\ominus < 0$, etc.).

Consider the running example again. The program at this stage is as follows:

$$\text{mg}(P, T, R, B) :- T = 0, B = P.$$

$$\text{mg}(P, T, R, B) :- P \geq 0, P1 = P * 1.01 - R, \text{mg}(P1, T1, R, B), T1 = T - 1, T > 0.$$

The LSign analysis produces the description $\{\{\langle \oplus = \oplus T, \text{One} \rangle\}\}$ (when projected on the variable T) for the program point just before the constraint $T > 0$. Using the equation $\oplus = \oplus T$ to simplify the constraint $T > 0$ gives $\oplus > 0$ which is trivially satisfiable. This shows that the constraint is redundant and can therefore be removed from the program.

1.7.3 Refinement Phase

After constraint removal, the refinement phase considers all constraints in all clauses and specializes them whenever possible. It perform both a Prop analysis and an LSign analysis on the program obtained after reordering and removal. The specialization of inequalities only uses the results of Prop and produces a test whenever all variables are fixed. The specialization of equations also uses the results of LSign to determine unconstrained variables. Consider the running example during this phase:

$\text{mg}(P, T, R, B) :- T = 0, B = P.$

$\text{mg}(P, T, R, B) :- P \geq 0, P1 = P * 1.01 - R, \text{mg}(P1, T1, R, B), T1 = T - 1.$

Given the top level input pattern, the input description for all calls to mg can be computed and it is given by $P \wedge R$ in the Prop domain and

$$\begin{aligned} &\{ \\ &\quad \{\langle T = \oplus P, \text{One} \rangle, \langle T = \oplus R, \text{One} \rangle\}, \\ &\quad \{\langle T = \oplus R, \text{One} \rangle, \langle T = \oplus P + \oplus R, \text{One} \rangle\}, \\ &\quad \{\langle T = \oplus R, \text{One} \rangle, \langle T \leq \ominus R, \text{One} \rangle, \langle T \leq \ominus R, \text{Any} \rangle, \langle T = \oplus P + \oplus R, \text{One} \rangle\} \\ &\} \end{aligned}$$

in the LSign domain. In the second clause, the first inequality is transformed into a test, since P is fixed in the input description of mg . Since R is also fixed and since $P1$ is unconstrained, the second constraint can be transformed into an assignment. Similar reasoning can be applied for the remaining constraints to obtain the final program:

$\text{mg}(P, T, R, B) :- T := 0, B := P.$

$\text{mg}(P, T, R, B) :- P \text{ ?} \geq 0, P1 := P * 1.01 - R, \text{mg}(P1, T1, R, B), T := T1 + 1.$

1.8 Summary of Contributions

The contributions of this thesis may be broadly divided into four categories. The first two sets of results pertain to the theory underlying the abstract domains LSign and

LInt, which are used in the analyses for reordering, refinement and removal. The third set of results deals with the theory underlying the reordering optimization. The final set of results is experimental, indicating that our approach to CLP optimization is promising and can yield substantial speedups in program execution.

The Domain LSign The thesis contributes to the theory underlying the abstract domain LSign [37, 43]. LSign is an elegant domain for analyzing CLP languages over linear real constraints. Its key conceptual ideas are the abstraction of linear constraints by replacing coefficients by signs and the use of annotations for preserving multiplicity information on the constraints. Unfortunately, the original paper on LSign by Marriott and Stuckey [37] has a number of theoretical drawbacks. In particular, the ordering of abstract constraint stores given in [37] does not capture the intended meaning and makes it impossible to prove the consistency of the abstract operations of LSign. This thesis reconsiders the domain LSign, and corrects and completes the results of [37]. Our main contributions in this respect are summarized here. The thesis gives the definition of ordering and a polynomial time algorithm for its implementation. It also gives an upper bound operation for the domain, proposes a simpler and more precise algorithm for abstract projection. and gives algorithms for satisfiability, redundancy and freeness analyses using the domain. The thesis reports the first implementation of the domain LSign.

The Domain LInt The thesis also proposes the domain LInt which is a generalization of LSign, abstracting coefficients by intervals instead of signs. This enables a more accurate analysis of programs in many cases. The main technical difficulty is that LInt is an infinite domain unlike LSign, and therefore requires the definition of a widening operator to guarantee the termination of analyses. The implementation of LInt indicates that the domain can provide a practical analysis of programs and does not pay too much penalty in time of analysis as compared to LSign.

The Reordering Optimization The thesis is the first work to formally specify what it means for a reordering optimization on CLP programs to be correct and it is also the first to present (with proof of correctness), a sufficient condition for performing correct reordering optimizations of $\text{CLP}(\mathcal{R}_{Lin})$ programs. The correctness criterion essentially guarantees that for any finite execution of the original program, the reordered program must mimic it. The thesis also shows how the correctness criterion may be reduced to a

satisfiability problem on constraint stores which can then be answered (in a conservative fashion) using the domains `LSign` or `LInt`.

Optimizing Compiler for $\text{CLP}(\mathcal{R}_{Lin})$ The primary aim of this thesis is to implement an optimizing compiler for $\text{CLP}(\mathcal{R}_{Lin})$, and we have succeeded in doing so. The experimental results obtained (summarized in Table 1.1) indicate the promise of our approach. Our main contributions in this respect are summarized here. The thesis presents the first provably correct implementation of reordering as well as the first implementation of constraint removal for $\text{CLP}(\mathcal{R}_{Lin})$ programs. Along with the implementation of constraint refinement, this produces the first $\text{CLP}(\mathcal{R}_{Lin})$ compiler integrating all these source to source transformations. Table 1.1 gives the speedup (ratio of execution time for unoptimized program to execution time for optimized program) observed for a variety of benchmarks. The experimental results show that sophisticated static analyses and source to source transformations can produce dramatic speedups for $\text{CLP}(\mathcal{R}_{Lin})$ programs, indicating the promise of our approach. In particular, as evidenced by the example `Triangular`, the speedups can increase as the size of the inputs increases. Also, a \checkmark in the reordering column indicates that the optimized program was reordered w.r.t. the original program. Reordering is seen to be the most powerful optimization, producing the most dramatic speedups. This is because reordering a program often leads to a complete bypass of the constraint solver (indicated by a \checkmark in the bypass column). In other words, the unoptimized program utilizes the constraint solver, while the optimized program does not. In that case, the unoptimized program performs costly constraint solving, while the optimized program performs only tests and assignments which are cheaper to implement. While there are several issues open in making the implementation of the optimizations fast, the optimization times for benchmarks (given in milliseconds) indicate that the analyses used are practical.

1.9 Organization of the Thesis

The rest of this thesis is organized in the following way. Chapter 2 gives a brief introduction to abstract interpretation, as well as the abstract interpretation algorithms and framework used in the compiler. The next three chapters present the various abstract domains used in the compiler. Chapters 3 and 4 give a detailed presentation of our work on the domains `LSign` and `LInt`, while chapter 5 reviews the domain `Prop`, used in

Program	Description	Speedup	Reorders?	Bypass?	Opt. Time
Integer	Is 25000 an integer?	1.08	×	×	150
	Generate 0 ... 250	17.18	✓	✓	420
Exp	Compute 2^{25}	1.00	✓	✓	530
	Compute $\lg 2^{25}$	10.00	✓	✓	710
	Generate (0, 1) ... (25, 2^{25})	12.00	✓	✓	1010
Sum	Compute $0 + 1 + \dots + 500$	5.00	✓	✓	1220
	Find N s.t. $0 + 1 + \dots + N = 125250$	1.11	✓	×	1510
	Generate (0, 0) ... (500, 125250)	14.44	✓	✓	1550
Fibonacci	Compute 15 th Fibonacci number	3.00	✓	✓	4330
	N s.t. 987 is N^{th} Fibonacci number	4.39	✓	×	8070
	Generate (0, 1) ... (15, 987)	9.73	✓	×	6270
Mortgage (Linear)	$P = 100, T = 50$; find B	1.05	×	×	1460
	$P = 200, T = 100$; find B	1.50	×	×	1460
	$P = 100, T = 0 \dots 50$; find B	2.10	✓	✓	3100
	$P = 200, T = 0 \dots 100$; find B	12.43	✓	✓	3100
Mortgage (Nonlinear)	$P = 100, T = 50$; find B	1.16	×	×	3440
	$P = 200, T = 100$; find B	1.25	×	×	3440
	$P = 100, T = 0 \dots 50$; find B	1.59	✓	✓	9560
	$P = 200, T = 0 \dots 100$; find B	11.36	✓	✓	9560
Ode-Euler	Compute final y value	1.13	×	×	1730
	Compute initial y value	1.07	✓	×	1700
	Relate initial and final y values	1.10	✓	×	1180
Triangular	Solve 2000 equations	4.00	✓	✓	390
	Solve 4000 equations	7.90	✓	✓	520
	Solve 8000 equations	15.48	✓	✓	2450
Ar. Mean		5.68			

Table 1.1: Experimental Results: Optimizing Compiler for $\text{CLP}(\mathcal{R}_{Lin})$.

groundness analysis. Chapter 6 presents the program transformations (refinement, removal and reordering) in greater detail, and formalizes the reordering optimization and its proof of correctness. Our experimental results and some details of the implementation are covered in chapter 7. We discuss the related research in chapter 8. We conclude in chapter 9 by reviewing our contributions and indicating some of the open issues in the area. Appendix A contains the details of various proofs in the thesis.

Chapter 2

Abstract Interpretation

In this chapter, we give an introduction to abstract interpretation and review the abstract interpretation framework used in the thesis. The presentation closely follows that of [29]. The chapter is organized in the following way. Section 2.1 introduces the basic ideas of abstract interpretation by means of an example. Section 2.2 presents the concrete semantics of CLP, while section 2.3 presents the abstract semantics of CLP.

2.1 Introduction

Abstract interpretation [10] 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;
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.

To illustrate these concepts, we use a small language of expressions built from the digits 0 to 9, juxtaposition of digits to obtain natural numbers, the operations $+$ and $-$ and a single variable whose name is x . The presentation is informal, but readers should have no difficulty in formalizing the concepts precisely.

A semantics for the language could be defined by a function $S : \text{Expression} \rightarrow \text{Integer} \rightarrow \text{Integer}$ as follows.

$$\begin{aligned}
S [E_1 + E_2] v &= S [E_1] v + S [E_2] v \\
S [E_1 - E_2] v &= S [E_1] v - S [E_2] v \\
S [0] v &= 0 \\
S [1] v &= 1 \\
&\vdots \\
S [9] v &= 9 \\
S [DN] v &= 10 \times S [D] v + S [N] v \\
S [x] v &= v
\end{aligned}$$

The semantics applied to an expression $x + 3 - x$ and a value 4 produces the value 3, i.e

$$S [x + 3 - x] 4 = 3.$$

The semantics can be generalized to a collecting (or concrete) semantics to produce results for a collection of inputs. The collecting semantics can be defined with various degrees of accuracy: it may contain only the correct results or it may contain the correct results and some additional elements. As a consequence, the collecting semantics may already make some approximation. For instance, a simple but not precise approximation of the semantics is given by the function $S_c : \text{Expression} \rightarrow 2^{\text{Integer}} \rightarrow 2^{\text{Integer}}$ defined as follows.

$$\begin{aligned}
S_c [E_1 + E_2] V &= S_c [E_1] V + S_c [E_2] V \\
S_c [E_1 - E_2] V &= S_c [E_1] V - S_c [E_2] V \\
S_c [0] V &= \{0\} \\
S_c [1] V &= \{1\} \\
&\vdots \\
S_c [9] V &= \{9\} \\
S_c [DN] V &= \{10\} \times S_c [D] V + S_c [N] V \\
S_c [x] V &= V
\end{aligned}$$

In this semantics, the operators $+$, $-$, \times have been overloaded to work on sets, i.e.

$$\{a_1, \dots, a_n\} + \{b_1, \dots, b_m\} = \{a_i + b_j \mid 1 \leq i \leq n \wedge 1 \leq j \leq m\}.$$

The collecting semantics applied to an expression $x + 3 - x$ and value $\{4\}$ produces the value $\{3\}$, i.e

$$S_c [x + 3 - x] \{4\} = \{3\}.$$

but the collecting semantics applied to $x + 3 - x$ and the value $\{4, 5\}$ produces the value $\{2, 3, 4\}$ since

$$\begin{aligned}
S_c [x + 3 - x] \{4, 5\} &= S_c [x + 3] \{4, 5\} - S_c [x] \{4, 5\} \\
&= S_c [x] \{4, 5\} + S_c [3] \{4, 5\} - S_c [x] \{4, 5\} \\
&= \{4, 5\} + \{3\} - \{4, 5\} \\
&= \{7, 8\} - \{4, 5\} \\
&= \{2, 3, 4\}
\end{aligned}$$

The collecting semantics can then be transformed into an abstract semantics by replacing the collecting domain by an abstract domain and by replacing each collecting operation by an abstract operation. The link between the abstract domain and the collecting domain is usually expressed by a monotone concretization function which maps abstract objects into collecting objects (i.e. sets of concrete objects). In the case of our simple language, one may want to approximate integers by their signs and to use an abstract domain consisting of the elements $\{\oplus, \ominus, 0, \top\}$. The meaning of the abstract objects is given by

$$\begin{aligned}
Cc(\oplus) &= \{n \mid n \text{ is an integer greater than } 0\} \\
Cc(\ominus) &= \{n \mid n \text{ is an integer smaller than } 0\} \\
Cc(0) &= \{0\} \\
Cc(\top) &= \{n \mid n \text{ is an integer}\}
\end{aligned}$$

This concretization is monotone if the abstract objects are ordered by using the rule

$$s_1 \leq s_2 \text{ if } s_1 = s_2 \vee s_2 = \top.$$

The next step of the abstraction process consists of defining abstract operations which safely approximate the collection operations. In our example, the operation $+$ must be approximated by an abstract operation $+$ which must satisfy the consistency condition

$$Cc(s_1) + Cc(s_2) \subseteq Cc(s_1 + s_2)$$

or equivalently

$$\forall n_1 \in Cc(s_1) \forall n_2 \in Cc(s_2) : n_1 + n_2 \in Cc(s_1 + s_2).$$

The abstract semantics can then be defined simply by replacing the collecting objects by their abstraction and the collecting operations by their abstract counterparts to obtain a function $S_a : \text{Expression} \rightarrow \text{Sign} \rightarrow \text{Sign}$ as follows:

$$\begin{aligned}
S_a [E_1 + E_2] s &= S_a [E_1] s + S_a [E_2] s \\
S_a [E_1 - E_2] s &= S_a [E_1] s - S_a [E_2] s \\
S_a [0] s &= 0 \\
S_a [1] s &= \oplus \\
&\vdots \\
S_a [9] s &= \oplus \\
S_a [DN] s &= \oplus \times S_a [D] s + S_a [N] s \\
S_a [x] s &= s
\end{aligned}$$

The evaluation of the abstract semantics on expression $x + 3 - x$ and sign \oplus produces

$$\begin{aligned}
S_a [x + 3 - x] \oplus &= S_c [x + 3] \oplus - S_c [x] \oplus \\
&= S_c [x] \oplus + S_c [3] \oplus - S_c [x] \oplus \\
&= \oplus + \oplus - \oplus \\
&= \oplus - \oplus \\
&= \top
\end{aligned}$$

which is not particularly accurate.

The last step of the methodology consists in applying an algorithm to compute the abstract semantics (or a sufficient subset of the abstract semantics to answer the original query). Since the semantic equations are generally mutually recursive, a fixpoint algorithm must be used for this purpose.

It is important to point out that there are many issues in designing an abstraction, some of which should be clear from our example. These issues include

1. how to define the semantics to capture the appropriate properties ?
2. how to define the collecting semantics ?
3. how to define an abstraction of the collecting semantics which is a good tradeoff between accuracy and efficiency ?
4. how to implement the fixpoint algorithm ?

Most of the results on abstract interpretation in this thesis deal with the third issue, and we use standard results for (1), (2) and (4). Note also that there are other approaches to abstract interpretation, e.g. using abstraction functions instead of concretization functions. Note finally that in this thesis we use the terms concrete and collecting semantics interchangeably.

2.2 Concrete Semantics

In this section, we give the concrete semantics for CLP, which is a collecting fixpoint semantics for the programming language. The concrete semantics is a natural extension to CLP of the logic programming semantics presented in [29]. 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 the section is organized as follows. Section 2.2.1 defines normalized $\text{CLP}(\mathcal{R}_{Lin})$ programs. The concrete objects are presented in Section 2.2.2 followed by the operations of the concrete semantics in Section 2.2.3. The transformation that gives us the concrete semantics is presented in Section 2.2.4.

2.2.1 Normalized Programs

It is convenient to restrict the syntax of $\text{CLP}(\mathcal{R}_{Lin})$ programs in order to simplify the analysis. The syntax of programs can be restricted by requiring that all programs be normalized. In order to define normalized programs, we assume the existence of sets P_i ($i \geq 0$) representing the set of predicate symbols of arity i , and an infinite sequence of variables x_1, x_2, \dots . Also, in the rest of the thesis, we do not permit functors to occur in $\text{CLP}(\mathcal{R}_{Lin})$ programs.

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 $p(x_1, \dots, x_n)$ 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 $p(x_{i_1}, \dots, x_{i_m})$ where $p \in P_m$ and the variables x_{i_1}, \dots, x_{i_m} are all distinct. A linear constraint has the form $c_0 \delta c_1 x_{i_1} + \dots + c_m x_{i_m}$ where $\delta \in \{=, <, \leq, >, \geq, \neq\}$ and each c_i is a rational number.

It is not a difficult matter to transform any given $\text{CLP}(\mathcal{R}_{Lin})$ program into an equivalent normalized program by simple rewriting rules. The use of normalized programs, first proposed for logic programs in [4], simplifies the semantics.

2.2.2 Concrete Objects

The concrete objects manipulated by our concrete semantics are linear constraints and multisets of linear constraints. Consider a set of variables $D = \{x_1, \dots, x_n\}$. 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. We denote multiset union by \sqcup . In the following, we denote 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 . We denote the set of variables $\{x_1, \dots, x_n\}$ by D_n for any n . If θ is a constraint store over the set of variables D , we denote D as $\text{dom}(\theta)$. The projection of a constraint store θ on the set of variables D is denoted $\theta|_D$.

2.2.3 Concrete Operations

The following operations are necessary to define the concrete semantics. The operations are described informally, followed by a formal specification.

Upper bound Let $\Theta_1, \dots, \Theta_m$ be sets of constraint stores on D_n . $\text{UNION}(\Theta_1, \dots, \Theta_m)$ returns a set of constraint stores that represents all the constraint stores represented by any 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 $\Theta_1, \dots, \Theta_m$ be sets of constraint stores on D_n . Then

$$\text{UNION}(\Theta_1, \dots, \Theta_m) = \Theta_1 \cup \dots \cup \Theta_m.$$

Addition of a constraint Let Θ be a set of constraint stores on D_n and λ be a constraint on D_n . $\text{AI_ADD}(\lambda, \Theta)$ 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

$$\text{AI_ADD}(\lambda, \Theta) = \{\theta \sqcup \{\lambda\} \mid \theta \in \Theta\}.$$

Restriction of a clause substitution Let Θ be a set of constraint stores on the clause variables D_m of a clause c and let D_n be the head variables of the clause ($n \leq m$). $\text{RESTRC}(c, \Theta)$ returns the set of constraint stores obtained by projecting Θ on the head variables. Operation RESTRC is used at the end of a clause execution to restrict the stores to the head variables. It is specified as follows.

Specification 3 Let Θ be a set of constraint on D_m (the variables of clause c) and let D_n be the head variables of c ($n \leq m$). Then

$$\text{RESTRC}(c, \Theta) = \{\theta_{/D_n} \mid \theta \in \Theta\}.$$

Extension of a clause substitution Let Θ be a set of constraint stores on the head variables D_n of a clause c , and let c contain the variables D_m ($n \leq m$). $\text{EXTC}(c, \Theta)$ returns the set of constraint stores obtained by extending Θ to accommodate the additional

new variables $\{x_{n+1}, \dots, x_m\}$ in the body of the clause. Operation EXTC is used at the beginning of a clause execution to express the input stores on all the clause variables and not just the head variables. It is specified as follows.

Specification 4 Let Θ be a set of constraint stores on D_n (the head variables of clause c) and let D_m be the clause variables of c ($n \leq m$). Then

$$\text{EXTC}(c, \Theta) = \{\theta \mid \text{dom}(\theta) = D_m \wedge \theta|_{D_n} \in \Theta \wedge \forall \lambda \in \theta \forall x_i \in D_m \setminus D_n : \lambda[i] = 0\}.$$

Restriction of a substitution before a literal Let Θ be a set of constraint stores on the variables D_n , and let l be a literal on the variables $\{x_{i_1}, \dots, x_{i_m}\}$ (where the variables appear in the literal in that order). $\text{RESTRG}(l, \Theta)$ returns the set of constraint stores obtained by

1. projecting Θ on $\{x_{i_1}, \dots, x_{i_m}\}$ obtaining Θ_1 ;
2. expressing Θ_1 on $\{x_1, \dots, x_m\}$ by mapping x_{i_k} to x_k .

Operation RESTRG is used before the execution of a literal in the body of a clause. Before specifying it formally, it is necessary to assume a renaming operation called the normalization. $\text{norm}[x_{i_1}, \dots, x_{i_m}] \Theta$ is a set of constraint stores obtained by simultaneously replacing each x_{i_j} by x_j in Θ . Operation RESTRG is specified as follows.

Specification 5 Let Θ be a set of constraint stores on D_n and let l be a literal on the variables $D_l = \{x_{i_1}, \dots, x_{i_m}\}$ (where the variables appear in the literal in that order). Let $D_l \subseteq D_n$. Then $\text{RESTRG}(l, \Theta) = \text{norm}[x_{i_1}, \dots, x_{i_m}] \Theta_1$ where

$$\Theta_1 = \{\theta|_{\{x_{i_1}, \dots, x_{i_m}\}} \mid \theta \in \Theta\}.$$

Extension of a substitution after a literal Let Θ be a set of constraint stores on D_n , and let Θ' be a set of constraint stores on D_m , representing the result of executing the literal l (in which the variables x_{i_1}, \dots, x_{i_m} occur in that order) with input $\text{RESTRG}(l, \Theta)$. $\text{EXTG}(l, \Theta, \Theta')$ returns a set of constraint stores which instantiates Θ to take into account the result Θ' of the literal l . It does this by

1. expressing Θ' on $\{x_{i_1}, \dots, x_{i_m}\}$ by mapping x_j to x_{i_j} ;
2. combining the above with Θ to reflect the result of executing l .

Operation EXTG is used after the execution of a literal in a clause to extend its result to the clause's store. Before specifying it formally, it is necessary to assume a renaming operation called the denormalization. $\text{denorm}[x_{i_1}, \dots, x_{i_m}] \Theta$ is a set of constraint stores obtained by simultaneously replacing each x_j by x_{i_j} in Θ . Operation EXTG is specified as follows.

Specification 6 Let Θ be a set of constraint stores on D_n , and let Θ' be a set of constraint stores on D_m . Let l be a literal on the variables $D_l = \{x_{i_1}, \dots, x_{i_m}\}$, in which the variables appear exactly in that order, and let $D_l \subseteq D_n$. Then

$$\text{EXTG}(l, \Theta, \Theta') = \{\theta \sqcup \theta_1 \mid \theta \in \Theta \wedge \theta_1 \in \Theta_1 \wedge \Theta_1 = \text{denorm}[x_{i_1}, \dots, x_{i_m}] \Theta'\}.$$

2.2.4 Concrete Transformation

We are now in position to present the concrete semantics. The concrete semantics is defined in terms of tuples of the form $(\Theta_{in}, p, \langle \Theta_{out}, \Theta_{int} \rangle)$ where p is a predicate symbol of arity n and $\Theta_{in}, \Theta_{out}$ and Θ_{int} are sets of constraint stores on the variables $\{x_1, \dots, x_n\}$. The informal reading of a tuple $(\Theta_{in}, p, \langle \Theta_{out}, \Theta_{int} \rangle)$ is as follows:

for every computation beginning at $\langle p(x_1, \dots, x_n) \sqcap \theta_{in} \rangle$, where $\theta_{in} \in \Theta_{in}$:

1. for every terminal computation state $\langle \epsilon \sqcap \theta_{out} \rangle$: $\theta_{out}/\{x_1, \dots, x_n\} \in \Theta_{out}$;
2. for every intermediate computation state $\langle G \sqcap \theta_{int} \rangle$: $\theta_{int}/\{x_1, \dots, x_n\} \in \Theta_{int}$.

Intuitively, Θ_{out} is the set of all possible output constraint stores (projected on the initial query variables) when p is executed with any store from Θ_{in} as the input. Θ_{int} is the set of all possible intermediate constraint stores (projected on the initial query variables), i.e. accumulated constraint store for any intermediate computation state, when p is executed with any store from Θ_{in} as the input. The concrete semantics is similar to that for logic programs in [29], except that it has a component for the intermediate descriptions in addition to the output descriptions.

We now present the transformation that gives us the concrete semantics. To simplify the presentation, the transformation is presented in two stages. First we present a transformation for a simplified concrete semantics with only the output descriptions. Its augmentation to include the intermediate descriptions is presented later.

$$TSCT(sct) = \{(\Theta_{in}, p, \Theta_{out}) \mid (\Theta_{in}, p) \in UD \wedge \Theta_{out} = \Gamma_p(\Theta_{in}, p, sct)\}$$

$$\begin{aligned} \Gamma_p(\Theta_{in}, p, sct) &= \text{UNION}(\Theta_{out}^1, \dots, \Theta_{out}^n) \\ \text{where } \Theta_{out}^i &= \Gamma_c(\Theta_{in}, c_i, sct) \quad c_1, \dots, c_n \text{ are the clauses of } p \end{aligned}$$

$$\begin{aligned} \Gamma_c(\Theta_{in}, c, sct) &= \text{RESTRC}(c, \Theta_{out}) \\ \text{where } \Theta_{out} &= \Gamma_b(\text{EXTC}(c, \Theta_{in}), b, sct) \quad b \text{ is the body of } c \end{aligned}$$

$$\begin{aligned} \Gamma_b(\Theta_{in}, <, >, sct) &= \Theta_{in} \\ \Gamma_b(\Theta_{in}, l.g, sct) &= \Gamma_b(\Theta_3, g, sct) \\ \text{where } \Theta_3 &= \text{EXTG}(l, \Theta_{in}, \Theta_2), \\ \Theta_2 &= \text{AI_ADD}(\lambda, \Theta_1) \quad \text{if } l \text{ is } \lambda, \text{ a constraint} \\ \Theta_1 &= \text{RESTRG}(l, \Theta_{in}) \end{aligned}$$

$$\begin{aligned} \Gamma_b(\Theta_{in}, l.g, sct) &= \Gamma_b(\Theta_3, g, sct) \\ \text{where } \Theta_3 &= \text{EXTG}(l, \Theta_{in}, \Theta_2), \\ \Theta_2 &= sct(\Theta_1, p) \quad \text{if } l \text{ is } p(\dots), \text{ a predicate} \\ \Theta_1 &= \text{RESTRG}(l, \Theta_{in}) \end{aligned}$$

Figure 2.1: Simplified Concrete Semantics

We denote by UD the underlying domain of the program, i.e. the set of pairs (Θ_{in}, p) where p is a predicate symbol of arity n and Θ_{in} is a set of constraint stores on the variables $\{x_1, \dots, x_n\}$. The simplified concrete semantics (without intermediate descriptions) is defined as the least fixpoint of the transformation $TSCT$ given in Figure 2.1. In the construction, sct is a set of concrete tuples and is functional, i.e. there exists at most one set of constraint stores Θ_{out} for each pair (Θ_{in}, p) such that $(\Theta_{in}, p, \Theta_{out}) \in sct$. We denote that set of constraint stores by $sct(\Theta_{in}, p)$.

Informally, the function $\Gamma_p(\Theta_{in}, p, sct)$ executes all the clauses in the definition of the predicate p for the input Θ_{in} and takes the upper bound of the results. Its result represents the output stores of the predicate. The function $\Gamma_c(\Theta_{in}, c, sct)$ executes a clause by extending the store to all the variables of the clause, executing the body, and restricting the stores to the head variables. It returns a set of stores representing the output for the clause. The function $\Gamma_b(\Theta_{in}, G, sct)$ executes the body of a clause by considering each literal in turn. For each literal, the set of stores is first expressed on the literal variables, then the literal is executed and then the result extended to the clause variables. If the literal is a predicate, the result is looked up in sct . Otherwise the literal is a constraint and operation AI_ADD is used.

$$TSCT(sct) = \{(\Theta_{in}, p, \langle \Theta_{out}, \Theta_{int} \rangle \mid (\Theta_{in}, p) \in UD \wedge \langle \Theta_{out}, \Theta_{int} \rangle = \Gamma_p(\Theta_{in}, p, sct)\}$$

$$\Gamma_p(\Theta_{in}, p, sct) = \langle \text{UNION}(\Theta_{out}^1, \dots, \Theta_{out}^n), \text{UNION}(\Theta_{int}^1, \dots, \Theta_{int}^n) \rangle$$

where $\langle \Theta_{out}^i, \Theta_{int}^i \rangle = \Gamma_c(\Theta_{in}, c_i, sct)$ c_1, \dots, c_n are the clauses of p

$$\Gamma_c(\Theta_{in}, c, sct) = \langle \text{RESTRC}(c, \Theta_{out}), \text{RESTRC}(c, \Theta_{int}) \rangle$$

where $\langle \Theta_{out}, \Theta_{int} \rangle = \Gamma_b(\text{EXTC}(c, \Theta_{in}), \text{EXTC}(c, \Theta_{in}), b, sct)$ b is the body of c

$$\Gamma_b(\Theta_{in}, \Theta_{int}, <, >, sct) = \langle \Theta_{in}, \Theta_{int} \rangle$$

$$\Gamma_b(\Theta_{in}, \Theta_{int}, l.g, sct) = \Gamma_b(\Theta_3, \text{UNION}(\Theta_{int}, \Theta_3), g, sct)$$

where $\Theta_3 = \text{EXTG}(l, \Theta_{in}, \Theta_2)$,
 $\Theta_2 = \text{AI_ADD}(\lambda, \Theta_1)$ if l is λ , a constraint
 $\Theta_1 = \text{RESTRG}(l, \Theta_{in})$

$$\Gamma_b(\Theta_{in}, \Theta_{int}, l.g, sct) = \Gamma_b(\Theta_3, \text{UNION}(\Theta_{int}, \Theta_3, \Theta'_3), g, sct)$$

where $\Theta_3 = \text{EXTG}(l, \Theta_{in}, \Theta_2)$,
 $\Theta'_3 = \text{EXTG}(l, \Theta_{in}, \Theta'_2)$,
 $\langle \Theta_2, \Theta'_2 \rangle = sct(\Theta_1, p)$ if l is $p(\dots)$, a predicate
 $\Theta_1 = \text{RESTRG}(l, \Theta_{in})$

Figure 2.2: Concrete Semantics

We now indicate how to augment the simplified concrete semantics to compute the intermediate stores as well. The key idea is to accumulate the intermediate stores while traversing the body of the clause. The concrete semantics is defined as the least fixpoint of the transformation $TSCT$ given in Figure 2.2. In the construction, sct is a set of concrete tuples and is functional, i.e. there exists at most one pair $\langle \Theta_{out}, \Theta_{int} \rangle$ for each pair (Θ_{in}, p) such that $(\Theta_{in}, p, \langle \Theta_{out}, \Theta_{int} \rangle) \in sct$. We denote that pair by $sct(\Theta_{in}, p)$.

Informally, the function $\Gamma_p(\Theta_{in}, p, sct)$ executes all the clauses in the definition of the predicate p for the input Θ_{in} and takes the upper bound of the results. It returns a pair, whose first element represents the output stores and whose second element represents the intermediate stores. The function $\Gamma_c(\Theta_{in}, c, sct)$ executes a clause by extending the store to all the variables of the clause, executing the body, and restricting the stores to the head variables. It also returns a $\langle output, intermediate \rangle$ set of stores pair. The function $\Gamma_b(\Theta_{in}, \Theta_{int}, G, sct)$ executes the body of a clause by considering each literal in turn. For each literal, the set of stores is first expressed on the literal variables, then the literal is executed and then the result extended to the clause variables. If the literal is a predicate, the result is looked up in sct . Otherwise the literal is a constraint

and operation `AI_ADD` is used. The function Γ_b also accumulates the intermediate stores of the computation. Initially the intermediate substitution is the same as the input substitution. As the body is traversed, the succeeding inputs (represented by Θ_3) are used to update the accumulated intermediate substitution by using the operation `UNION`. In the case of predicate calls in the body, the intermediate substitutions that can occur inside the predicate call (represented by Θ'_3) also need to be added to the accumulated intermediate substitution.

2.3 Abstract Semantics

The abstract semantics is a natural extension to CLP of the logic programming abstract semantics in [29], and is close to the works of [54] and [33]. The abstract semantics consists of approximating the concrete semantics by replacing a set of constraint stores by a single abstract substitution, i.e. an abstract substitution represents a set of constraint stores. Therefore, the abstract semantics is defined in terms of abstract tuples of the form $(\beta_{in}, p, \langle \beta_{out}, \beta_{int} \rangle)$ where p is a predicate symbol of arity n and β_{in}, β_{out} and β_{int} are abstract substitutions on the variables $\{x_1, \dots, x_n\}$. The informal reading of an abstract tuple $(\beta_{in}, p, \langle \beta_{out}, \beta_{int} \rangle)$ is as follows:

for every computation beginning at $\langle p(x_1, \dots, x_n) \sqcap \theta_{in} \rangle$, where θ_{in} is a constraint store satisfying the property expressed by β_{in} :

1. for every terminal computation state $\langle \epsilon \sqcap \theta_{out} \rangle$: $\theta_{out}/\{x_1, \dots, x_n\}$ is a constraint store that satisfies the property expressed by β_{out} ;
2. for every intermediate computation state $\langle G \sqcap \theta_{int} \rangle$: $\theta_{int}/\{x_1, \dots, x_n\}$ is a constraint store that satisfies the property expressed by β_{int} .

Intuitively, β_{out} represents all the constraint stores that can be the output, and β_{int} represents all the intermediate constraint stores. when p is executed with any store satisfying β_{in} as the input.

The section is organized as follows. First we present the abstract operations in Section 2.3.1, followed by the transformation for the abstract semantics in Section 2.3.2. Section 2.3.3 describes how the fixpoint of the abstract semantics may be computed.

2.3.1 Abstract Operations

The following operations are necessary to define the abstract semantics. The operations are described informally, followed by a formal specification.

Upper bound Let β_1, \dots, β_m be abstract substitutions on D_n . $\text{UNION}(\beta_1, \dots, \beta_m)$ returns an abstract substitution that represents all the constraint stores represented by any β_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 7 Let β_1, \dots, β_m be abstract substitutions on D_n . Let $\Theta_1 = Cc(\beta_1), \dots, \Theta_m = Cc(\beta_m)$ and $\Theta' = \text{UNION}(\Theta_1, \dots, \Theta_m)$. Then $\beta' = \text{UNION}(\beta_1, \dots, \beta_m)$ is an abstract substitution on D_n s.t. $\Theta' \subseteq Cc(\beta')$.

Addition of a constraint Let β be an abstract substitution on D_n and λ be a constraint on D_n . $\text{AI_ADD}(\lambda, \beta)$ returns an abstract substitution that represents the result of adding the constraint λ to any constraint store represented by β . Operation AI_ADD is used when a constraint is encountered in the program. It is specified as follows.

Specification 8 Let β be an abstract substitution on D_n and λ be a constraint on D_n . Let $\Theta = Cc(\beta)$ and $\Theta' = \text{AI_ADD}(\lambda, \Theta)$. Then $\beta' = \text{AI_ADD}(\lambda, \beta)$ is an abstract substitution on D_n s.t. $\Theta' \subseteq Cc(\beta')$.

Restriction of a clause substitution Let β be an abstract substitution on the clause variables D_m of a clause c and let D_n be the head variables of the clause ($n \leq m$). $\text{RESTRC}(c, \beta)$ returns the abstract substitution obtained by projecting β on the head variables. Operation RESTRC is used at the end of a clause execution to restrict the substitution to the head variables. It is specified as follows.

Specification 9 Let β be an abstract substitution on D_m (the variables of clause c) and let D_n be the head variables of c ($n \leq m$). Let $\Theta = Cc(\beta)$ and $\Theta' = \text{RESTRC}(c, \Theta)$. Then $\beta' = \text{RESTRC}(c, \beta)$ is an abstract substitution on D_n s.t. $\Theta' \subseteq Cc(\beta')$.

Extension of a clause substitution Let β be an abstract substitution on the head variables D_n of a clause c , and let c contain the variables D_m ($n \leq m$). $\text{EXTC}(c, \beta)$ returns the abstract substitution obtained by extending β to accomodate the additional

new variables $\{x_{n+1}, \dots, x_m\}$ in the body of the clause. Operation EXTC is used at the beginning of a clause execution to express the input substitution on all the clause variables and not just the head variables. It is specified as follows.

Specification 10 Let β be an abstract substitution on D_n (the head variables of clause c) and let D_m be the clause variables of c ($n \leq m$). Let $\Theta = Cc(\beta)$ and $\Theta' = \text{EXTC}(c, \Theta)$. Then $\beta' = \text{EXTC}(c, \beta)$ is an abstract substitution on D_m s.t. $\Theta' = Cc(\beta')$.

Restriction of a substitution before a literal Let β be an abstract substitution on the variables D_n , and let l be a literal on the variables $\{x_{i_1}, \dots, x_{i_m}\}$ (where the variables appear in the literal in that order). $\text{RESTRG}(l, \beta)$ returns the abstract substitution obtained by

1. projecting β on $\{x_{i_1}, \dots, x_{i_m}\}$ obtaining β_l ;
2. expressing β_l on $\{x_1, \dots, x_m\}$ by mapping x_{i_k} to x_k .

Operation RESTRG is used before the execution of a literal in the body of a clause. Operation RESTRG is specified as follows.

Specification 11 Let β be an abstract substitution on D_n and let l be a literal on the variables $D_l = \{x_{i_1}, \dots, x_{i_m}\}$ (where the variables appear in the literal in that order). Let $D_l \subseteq D_n$. Let $\Theta = Cc(\beta)$ and $\Theta' = \text{RESTRG}(l, \Theta)$. Then $\beta' = \text{RESTRG}(l, \beta)$ is an abstract substitution on D_m s.t. $\Theta' \subseteq Cc(\beta')$.

Extension of a substitution after a literal Let β be an abstract substitution on D_n , and let β' be an abstract substitution on D_m , representing the result of executing the literal l (in which the variables x_{i_1}, \dots, x_{i_m} occur, in that order) with input $\text{RESTRG}(l, \beta)$. $\text{EXTG}(l, \beta, \beta')$ returns an abstract substitution which instantiates β abstractly to take into account the result β' of the literal l . It does this by

1. expressing β' on $\{x_{i_1}, \dots, x_{i_m}\}$ by mapping x_j to x_{i_j} ;
2. combining the above with β to reflect the result of executing l .

Operation EXTG is used after the execution of a literal in a clause to extend its result to the clause substitution. Operation EXTG is specified as follows.

Specification 12 Let β be an abstract substitution on D_n , and let β' be an abstract substitution on D_m . Let l be a literal on the variables $D_l = \{x_{i_1}, \dots, x_{i_m}\}$, in which the variables appear exactly in that order, and let $D_l \subseteq D_n$. Let $\Theta = Cc(\beta)$, $\Theta' = Cc(\beta')$ and $\Theta'' = \text{EXTG}(l, \Theta, \Theta')$. Then $\beta'' = \text{EXTG}(l, \beta, \beta')$ is an abstract substitution on D_n s.t. $\Theta'' \subseteq Cc(\beta'')$.

2.3.2 Abstract Transformation

We are now in position to present the abstract semantics. Once again, it is convenient to present the semantics in two steps. First we present the semantics without the intermediate descriptions. In the second step, the semantics is augmented to compute the intermediate descriptions as well.

We denote by UD the underlying domain of the program, i.e. the set of pairs (β_{in}, p) where p is a predicate symbol of arity n and β_{in} is an abstract substitution on the variables $\{x_1, \dots, x_n\}$. The abstract semantics is defined as the least fixpoint of the transformation $TSAT$ given in Figure 2.3. In the construction, sat is a set of abstract tuples and is functional, i.e. there exists at most abstract substitution β_{out} for each pair (β_{in}, p) such that $(\beta_{in}, p, \beta_{out}) \in sat$. We denote that abstract substitution by $sat(\beta_{in}, p)$.

Informally, the function $\tau_p(\beta_{in}, p, sat)$ executes all the clauses in the definition of the predicate p for the input β_{in} and takes the upper bound of the results. It returns an abstract substitution representing the output of the predicate. The function $\tau_c(\beta_{in}, c, sat)$ executes a clause by extending the substitution to all the variables of the clause, executing the body, and restricting the substitution to the head variables. It also returns an abstract substitution. The function $\tau_b(\beta_{in}, G, sat)$ executes the body of a clause by considering each literal in turn. For each literal, the substitution is first expressed on the literal variables, then the literal is executed and then the result extended to the clause variables. If the literal is a predicate, the result is looked up in sat . Otherwise the literal is a constraint and operation `AI_ADD` is used.

We now indicate how to augment the simplified abstract semantics to compute the intermediate substitution as well.¹ The key idea is to accumulate the intermediate substitution while traversing the body of the clause. The abstract semantics is defined as the

¹ It may be possible to compute the intermediate stores using only the input and output stores and a post-processing step. However it is not obvious that the intermediate stores as computed by such a step would correspond with the intermediate stores of the concrete semantics.

$$TSAT(sat) = \{(\beta_{in}, p, \beta_{out}) \mid (\beta_{in}, p) \in UD \wedge \beta_{out} = \tau_p(\beta_{in}, p, sat)\}$$

$$\begin{aligned} \tau_p(\beta_{in}, p, sat) &= \text{UNION}(\beta_{out}^1, \dots, \beta_{out}^n) \\ \text{where } \beta_{out}^i &= \tau_c(\beta_{in}, c_i, sat) \quad c_1, \dots, c_n \text{ are the clauses of } p \end{aligned}$$

$$\begin{aligned} \tau_c(\beta_{in}, c, sat) &= \text{RESTRC}(c, \beta_{out}) \\ \text{where } \beta_{out} &= \tau_b(\text{EXTC}(c, \beta_{in}), b, sat) \quad b \text{ is the body of } c \end{aligned}$$

$$\begin{aligned} \tau_b(\beta_{in}, <, >, sat) &= \beta_{in} \\ \tau_b(\beta_{in}, l.g, sat) &= \tau_b(\beta_3, g, sat) \\ \text{where } \beta_3 &= \text{EXTG}(l, \beta_{in}, \beta_2), \\ \beta_2 &= \text{AI_ADD}(\lambda, \beta_1) \quad \text{if } l \text{ is } \lambda, \text{ a constraint} \\ \beta_1 &= \text{RESTRG}(l, \beta_{in}) \end{aligned}$$

$$\begin{aligned} \tau_b(\beta_{in}, l.g, sat) &= \tau_b(\beta_3, g, sat) \\ \text{where } \beta_3 &= \text{EXTG}(l, \beta_{in}, \beta_2), \\ \beta_2 &= sat(\beta_1, p) \quad \text{if } l \text{ is } p(\dots), \text{ a predicate} \\ \beta_1 &= \text{RESTRG}(l, \beta_{in}) \end{aligned}$$

Figure 2.3: Simplified Abstract Semantics

least fixpoint of the transformation $TSAT$ given in Figure 2.4. In the construction, sat is a set of abstract tuples and is functional, i.e. there exists at most one pair $\langle \beta_{out}, \beta_{int} \rangle$ for each pair (β_{in}, p) such that $(\beta_{in}, p, \langle \beta_{out}, \beta_{int} \rangle) \in sat$. We denote that pair by $sat(\beta_{in}, p)$.

Informally, the function $\tau_p(\beta_{in}, p, sat)$ executes all the clauses in the definition of the predicate p for the input β_{in} and takes the upper bound of the results. It returns a pair, whose first element represents the output substitution and whose second element represents the intermediate substitution. The function $\tau_c(\beta_{in}, c, sat)$ executes a clause by extending the substitution to all the variables of the clause, executing the body, and restricting the substitution to the head variables. It also returns a $\langle output, intermediate \rangle$ abstract substitution pair. The function $\tau_b(\beta_{in}, \beta_{int}, G, sat)$ executes the body of a clause by considering each literal in turn. For each literal, the substitution is first expressed on the literal variables, then the literal is executed and then the result extended to the clause variables. If the literal is a predicate, the result is looked up in sat . Otherwise the literal is a constraint and operation AI_ADD is used. The function τ_b also accumulates the intermediate stores of the computation. Initially the intermediate substitution is the same as the input substitution. As the body is traversed, the succeeding inputs

$$TSAT(sat) = \{(\beta_{in}, p, \langle \beta_{out}, \beta_{int} \rangle \mid (\beta_{in}, p) \in UD \wedge \langle \beta_{out}, \beta_{int} \rangle = \tau_p(\beta_{in}, p, sat))\}$$

$$\begin{aligned} \tau_p(\beta_{in}, p, sat) &= \langle \text{UNION}(\beta_{out}^1, \dots, \beta_{out}^n), \text{UNION}(\beta_{int}^1, \dots, \beta_{int}^n) \rangle \\ \text{where } \langle \beta_{out}^i, \beta_{int}^i \rangle &= \tau_c(\beta_{in}, c_i, sat) \quad c_1, \dots, c_n \text{ are the clauses of } p \end{aligned}$$

$$\begin{aligned} \tau_c(\beta_{in}, c, sat) &= \langle \text{RESTRC}(c, \beta_{out}), \text{RESTRC}(c, \beta_{int}) \rangle \\ \text{where } \langle \beta_{out}, \beta_{int} \rangle &= \tau_b(\text{EXTC}(c, \beta_{in}), \text{EXTC}(c, \beta_{in}), b, sat) \quad b \text{ is the body of } c \end{aligned}$$

$$\begin{aligned} \tau_b(\beta_{in}, \beta_{int}, <, >, sat) &= \langle \beta_{in}, \beta_{int} \rangle \\ \tau_b(\beta_{in}, \beta_{int}, l.g, sat) &= \tau_b(\beta_3, \text{UNION}(\beta_{int}, \beta_3), g, sat) \\ \text{where } \beta_3 &= \text{EXTG}(l, \beta_{in}, \beta_2), \\ \beta_2 &= \text{AI_ADD}(\lambda, \beta_1) \quad \text{if } l \text{ is } \lambda, \text{ a constraint} \\ \beta_1 &= \text{RESTRG}(l, \beta_{in}) \end{aligned}$$

$$\begin{aligned} \tau_b(\beta_{in}, \beta_{int}, l.g, sat) &= \tau_b(\beta_3, \text{UNION}(\beta_{int}, \beta_3, \beta'_3), g, sat) \\ \text{where } \beta_3 &= \text{EXTG}(l, \beta_{in}, \beta_2), \\ \beta'_3 &= \text{EXTG}(l, \beta_{in}, \beta'_2), \\ \langle \beta_2, \beta'_2 \rangle &= sat(\beta_1, p) \quad \text{if } l \text{ is } p(\dots), \text{ a predicate} \\ \beta_1 &= \text{RESTRG}(l, \beta_{in}) \end{aligned}$$

Figure 2.4: Abstract Semantics

(represented by β_3) are used to update the accumulated intermediate substitution by using the operation UNION. In the case of predicate calls in the body, the intermediate substitutions that can occur inside the predicate call (represented by β'_3) also need to be added to the accumulated intermediate substitution.

2.3.3 Fixpoint

The final step of the abstract interpretation methodology consists of computing a least fixpoint or post-fixpoint of the abstract semantics. This can be computed in a top-down or a bottom-up fashion. Generic abstract interpretation algorithms from logic programming, such as GAIA [29] or PLAI [40] can be adapted for the purpose.

From the abstract interpretation results, our optimizing compiler uses basically three abstract substitutions for each predicate p in the program. It uses the pair $\langle \beta_{out}, \beta_{int} \rangle = sat(\emptyset, p)$ which represent the output and intermediate substitutions when the predicate p is called with an empty constraint store. These are called the goal independent (or online) *output description* and *intermediate description* of p respectively. The analysis

also collects all the inputs seen during the abstract interpretation for the given query and produces an abstract substitution β_{in} that summarizes these. This is called the *input description* of p for the given query.

Chapter 3

Abstract Domain LSign

In this chapter, we present the abstract domain `LSign` which is useful to deduce a variety of interesting properties of programs, including satisfiability of constraint stores and redundancy of constraints. The chapter is organized as follows. Section 3.1 presents the concrete objects whose properties are sought to be abstracted. Section 3.2 presents the abstract domain used to approximate the concrete objects. Sections 3.3 and 3.4 contain the operations and applications of the domain. This is followed by a brief presentation of the power domain in Section 3.5. The definition of the operations of the abstract interpretation framework in terms of the domain operations is presented in Section 3.6. Section 3.7 presents a complete worked example of the information collected during program analysis with the domain `LSign`. The chapter concludes with a discussion of the differences between the domain presented here with the original domain `LSign` as presented by Marriott and Stuckey [37]. The proofs of most of the results in this chapter can be found in Appendix A, although we sometimes include a sketch of the proof for the main results in the text.

3.1 Concrete Objects

We review some of the concepts from the concrete semantics here. The concrete objects manipulated by our concrete semantics are linear constraints and multisets of linear constraints. Consider a set of variables $D = \{x_1, \dots, x_n\}$. 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 $\delta \in \{<, \leq, =\}$. In this chapter we assume that constraints with operators $>$ and \geq have been rewritten in terms of $<$ and \leq . Also we do not consider disequations (constraints with operator \neq)

initially. We show how to extend the domain to accomodate disequations in Section 3.5. 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 .¹ The set of constraint stores over D is denoted by CS_D and is ordered by standard multiset inclusion. We denote multiset union by \sqcup . In the following, we denote linear constraints by the letter λ and constraint stores by the letter θ , both possibly subscripted. If λ is a linear constraint $c_0 \delta \sum_{i=1}^n c_i x_i$, $\lambda[i]$ denotes the coefficient c_i and $op(\lambda)$ denotes the operator δ . If θ is a constraint store over $D = \{x_1, \dots, x_n\}$ 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 $\text{Var}(\theta)$ to denote the set of variables with non-zero coefficients in θ . We also denote by \mathbb{N} the set $\{i \mid x_i \in D\}$.

3.2 Abstract Objects and Concretization

In this section, we introduce the domain LSign . Although the presentation differs considerably from [37], the domain is in fact a slight generalization of the original domain, which 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, [37] 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 $D = \{x_1, \dots, x_n\}$. 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 $\text{Sign} = \{0, \oplus, \ominus, \top\}$. Sign is ordered by

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

The concretization function $Cc : \text{Sign} \rightarrow 2^{\mathbb{R}}$ is defined as

$$\begin{aligned} Cc(0) &= \{0\}; & Cc(\oplus) &= \{c \mid c \in \mathbb{R} \wedge c > 0\}; \\ Cc(\ominus) &= \{c \mid c \in \mathbb{R} \wedge c < 0\}; & Cc(\top) &= \mathbb{R}. \end{aligned}$$

¹ Using multisets instead of sets simplifies a number of technical details in the proofs of our results.

² This was motivated by previous work on sequences [28, 3] which separates properties of the elements of the sequences from properties of the sequence.

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 $\text{Op} = \{<, \leq, =\}$. 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

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

The concretization function $Cc : A_D \rightarrow C_D$ 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 4 The abstract constraint $\top = \oplus P + \oplus R$ represents both the constraint $3 = P + R$ and $-3 = 2P + 5R$ but not the constraint $3 = -P + R$.

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 $\text{Mult} = \{\text{One}, \text{ZeroOrOne}, \text{Any}\}$. Mult is ordered by $\text{One} \sqsubseteq \text{ZeroOrOne} \sqsubseteq \text{Any}$. Multiplicities are denoted by the letter μ , possibly subscripted.

³[37] contains one other multiplicity *OneOrMore* which is obtained easily in our domain by including one constraint with multiplicity *One* and one constraint with multiplicity *Any*. Note also that all inequalities are defined with a multiplicity *Any* in [37].

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 $Cc : AM_D \rightarrow CS_D$ is defined as

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

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 5 The abstract constraint with multiplicity $\langle \top = \oplus P + \oplus R, One \rangle$ represents only multisets of size 1, e.g., $\{3 = P + R\}$. $\langle 0 \leq \oplus P + \oplus R, Any \rangle$ represents multisets of any size, e.g., \emptyset , $\{0 \leq 3P - R\}$ and $\{0 \leq 3P - R, 0 \leq 2P - 3R\}$.

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.

$$\begin{aligned} Cc_i(\emptyset) &= \{ \emptyset \} \\ Cc_i(\{ \gamma \} \cup \beta) &= \{ \theta_1 \sqcup \theta_2 \mid \theta_1 \in Cc(\gamma) \wedge \theta_2 \in Cc_i(\beta) \} \quad \gamma \notin \beta. \end{aligned}$$

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

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 $\text{mult}(\gamma) = \text{One}$. γ is a *possible* abstract constraint iff $\text{mult}(\gamma) = \text{ZeroOrOne}$. It is an *indefinite* abstract constraint otherwise. The *definite portion* of β , denoted by $\text{Def}(\beta)$, is the set of definite constraints of β . The *possible portion* of β , denoted by $\text{Pos}(\beta)$, is the set of possible constraints of β . The *indefinite portion* of β , denoted by $\text{Indef}(\beta)$, 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. $\forall \gamma_1, \gamma_2 \in \beta_1: \gamma_1 \neq \gamma_2 \Rightarrow f(\gamma_1) \neq f(\gamma_2) \vee f(\gamma_1) \in \text{Indef}(\beta_2)$.
3. $\text{Def}(\beta_2) \subseteq \text{range}(f)$.

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.

Example 7 Consider the following abstract constraints with multiplicity.

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

$\{\gamma_1, \gamma_2\} \sqsubseteq \{\gamma_4, \gamma_5\}$ and the ordering function is defined by $f(\gamma_1) = \gamma_4$ and $f(\gamma_2) = \gamma_5$.

$\{\gamma_1, \gamma_2, \gamma_3\} \sqsubseteq \{\gamma_4, \gamma_5\}$ and the function is defined by $f(\gamma_1) = \gamma_4$, $f(\gamma_2) = \gamma_5$, and $f(\gamma_3) = \gamma_5$.

$\{\gamma_3, \gamma_4\} \not\sqsubseteq \{\gamma_1, \gamma_5, \gamma_6\}$, 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 chapter: 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). The lifting function is related to, but not exactly the same as the abstraction function used in the abstract interpretation literature.

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 $f : \theta \rightarrow \beta$ satisfying

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

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

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,

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

(\Rightarrow) Let $\theta \in Cc_i(\beta)$. Hence $\theta = \theta_1 \sqcup \theta'$, 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 $f : \theta \rightarrow \beta$ 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 $\forall \lambda \in \theta_1 : \lambda \in Cc(\text{cons}(\gamma))$ by Definition 11 and $\theta_1 \in Cc(\gamma)$ by Definition 6. Let $\theta = \theta_1 \sqcup \theta'$ and consider the function $f' : \theta' \rightarrow \beta'$ 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, $\theta_1 \sqcup \theta' \in Cc_i(\{\gamma\} \cup \beta')$.

□

Example 8 Consider the following abstract constraints with multiplicity

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

and concrete constraints

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

$\{\lambda_1, \lambda_2\} \in Cc_i(\{\gamma_2, \gamma_3\})$ 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 chapter.

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 , $\theta \in Cc(\beta_2)$. \square

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.

3.3 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.

3.3.1 Ordering

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

Definition 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 (with weights associated with the edges). 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 $\text{Def}(\beta_2)$, a vertex for each constraint in $\text{Pos}(\beta_2)$, and $|\beta_1|$ vertices for each constraint in $\text{Indef}(\beta_2)$, 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

$$\begin{aligned}
V &= V_1 \cup V_2 \cup V_3 \cup V_4 \\
V_1 &= \{\gamma_1 \mid \gamma_1 \in \beta_1\} \\
V_2 &= \{\gamma_2 \mid \gamma_2 \in \text{Def}(\beta_2)\} \\
V_3 &= \{\gamma_2 \mid \gamma_2 \in \text{Pos}(\beta_2)\} \\
V_4 &= \{\gamma_2^\gamma \mid \gamma_2 \in \text{Indef}(\beta_2) \wedge \gamma \in \beta_1\} \\
\\
E &= E_2 \cup E_3 \cup E_4 \\
E_2 &= \{(\gamma_1, \gamma_2) \mid \gamma_1 \in V_1 \wedge \gamma_2 \in V_2 \wedge \gamma_1 \sqsubseteq \gamma_2\} \\
E_3 &= \{(\gamma_1, \gamma_2) \mid \gamma_1 \in V_1 \wedge \gamma_2 \in V_3 \wedge \gamma_1 \sqsubseteq \gamma_2\} \\
E_4 &= \{(\gamma_1, \gamma_2^\gamma) \mid \gamma_1 \in V_1 \wedge \gamma_2^\gamma \in V_4 \wedge \gamma_1 \sqsubseteq \gamma_2\}
\end{aligned}$$

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

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 $|\beta_1| + |\text{Def}(\beta_2)|$.

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 $|\beta_1| + |\text{Def}(\beta_2)|$.

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,

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

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

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

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

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

Theorem 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 $|\beta_1| + |\text{Def}(\beta_2)|$ when testing $\beta_1 \sqsubseteq \beta_2$. \square

Example 9 Consider the abstract stores $\beta_1 = \{\gamma_1, \gamma_2\}$ and $\beta_2 = \{\gamma_3, \gamma_4\}$, where

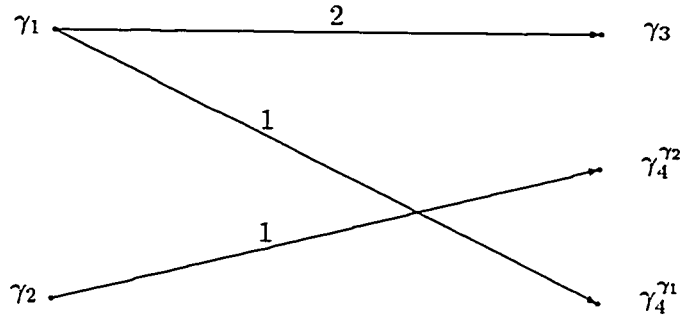
$$\gamma_1 = \langle \oplus = \oplus P + \top R, \text{One} \rangle$$

$$\gamma_2 = \langle \oplus = \oplus P + \ominus R, \text{Any} \rangle$$

$$\gamma_3 = \langle \oplus = \top P + \top R, \text{One} \rangle$$

$$\gamma_4 = \langle \oplus = \oplus P + \top R, \text{Any} \rangle.$$

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 [16] 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. \square

3.3.2 Addition

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

Specification 13 [Adding an Abstract Constraint with Multiplicity] $\uplus : AS_D \times AM_D \rightarrow AS_D$ should satisfy the following consistency condition. $\forall \theta_1, \theta_2 \in CS_D, \forall \beta \in AS_D, \forall \gamma \in AM_D$:

$$\theta_1 \in Cc(\beta) \wedge \theta_2 \in Cc(\gamma) \Rightarrow \theta_1 \sqcup \theta_2 \in Cc(\beta \uplus \gamma).$$

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.⁴

Example 10 Let $\beta = \{\langle \oplus = \oplus T, One \rangle, \langle 0 \leq \oplus P, One \rangle\}$. Adding the abstraction of $0 \leq 2R + 3B$, i.e. $\langle 0 \leq \oplus R + \oplus B, One \rangle$, to β gives the abstract store

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

Adding the abstraction of $0 \leq 3R + 4B$, i.e. $\langle 0 \leq \oplus R + \oplus B, One \rangle$, to β' gives the store

$$\{\langle \oplus = \oplus T, One \rangle, \langle 0 \leq \oplus P, One \rangle, \langle 0 \leq \oplus R + \oplus B, One \rangle, \langle 0 \leq \oplus R + \oplus B, Any \rangle\}.$$

Theorem 5 Let β be an abstract store and γ be an abstract constraint with multiplicity.

- (i) If $\theta_1 \in Cc_i(\beta)$ and $\theta_2 \in Cc(\gamma)$, then $\theta_1 \sqcup \theta_2 \in Cc_i(\beta \uplus \gamma)$.
- (ii) If $\theta_1 \in Cc(\beta)$ and $\theta_2 \in Cc(\gamma)$, then $\theta_1 \sqcup \theta_2 \in Cc(\beta \uplus \gamma)$.

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

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

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

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

The proof consists of showing that f' is a lifting function of $\theta_1 \sqcup \theta_2$ to $\beta \uplus \gamma$. □

Example 11 From the definition of the concretization,

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

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

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

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.

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

Lemma 5 Let β and β' be abstract stores.

(i) If $\theta \in Cc_i(\beta)$ and $\theta' \in Cc_i(\beta')$ then $\theta \sqcup \theta' \in Cc_i(\beta \uplus \beta')$.

(ii) If $\theta \in Cc(\beta)$ and $\theta' \in Cc(\beta')$ then $\theta \sqcup \theta' \in Cc(\beta \uplus \beta')$.

Example 12 The following example is based on the mortgage program `mg/4`. We have that $\theta \in Cc_i(\beta)$ where

$$\theta = \{ 0 < T, 0 \leq P, 0 = P * 1.01 - R - P1, 1 = T - T1 \}$$

and

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

Also $\theta' \in Cc_i(\beta')$ where

$$\theta' = \{ 0 = T1, 0 = B - P1 \}$$

and

$$\beta' = \{\langle 0 = \oplus T1, One \rangle, \langle 0 = \oplus B + \oplus P1, One \rangle\}.$$

This gives by Lemma 5 that $\theta \sqcup \theta' \in Cc(\beta \uplus \beta')$.

3.3.3 Upper Bound

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

Specification 14 $\text{UNION}: AS_D \times AS_D \rightarrow AS_D$ should satisfy the following consistency condition. $\forall \theta \in CS_D, \forall \beta_1 \in AS_D, \forall \beta_2 \in AS_D$:

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

We first define the upper bound of signs and multiplicities.

Definition 17 [Upper Bound on Signs] The upper bound operation on signs $\sqcup: \text{Sign} \times \text{Sign} \rightarrow \text{Sign}$ 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 $\sqcup: \text{Mult} \times \text{Mult} \rightarrow \text{Mult}$ 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 $\{\langle \oplus = \oplus x_1 + \oplus x_2, \text{One} \rangle\}$ and β_2 is $\{\langle \oplus = \oplus x_1 + \oplus x_2, \text{One} \rangle\}$, both

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

and

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

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 $\text{UNION}: AS_D \times AS_D \rightarrow AS_D$ is any operation obtained by applying in a

nondeterministic way the following set of rules.

- 1 $\text{UNION}(\emptyset, \emptyset) = \emptyset$
- 2 i $\text{UNION}(\{\langle \sigma, \text{One} \rangle\} \cup \beta'_1, \beta_2) = \text{UNION}(\beta'_1, \beta_2) \uplus \langle \sigma, \text{ZeroOrOne} \rangle$ if $\langle \sigma, \text{One} \rangle \notin \beta'_1$
ii $\text{UNION}(\beta_1, \{\langle \sigma, \text{One} \rangle\} \cup \beta'_2) = \text{UNION}(\beta_1, \beta'_2) \uplus \langle \sigma, \text{ZeroOrOne} \rangle$ if $\langle \sigma, \text{One} \rangle \notin \beta'_2$
- 3 i $\text{UNION}(\{\langle \sigma, \mu \rangle\} \cup \beta'_1, \beta_2) = \text{UNION}(\beta'_1, \beta_2) \uplus \langle \sigma, \mu \rangle$ if $\langle \sigma, \mu \rangle \notin \beta'_1 \wedge \mu \neq \text{One}$
ii $\text{UNION}(\beta_1, \{\langle \sigma, \mu \rangle\} \cup \beta'_2) = \text{UNION}(\beta_1, \beta'_2) \uplus \langle \sigma, \mu \rangle$ if $\langle \sigma, \mu \rangle \notin \beta'_2 \wedge \mu \neq \text{One}$
- 4 $\text{UNION}(\{\langle s_0 \delta \sum_{i=1}^n s_i x_i, \mu \rangle\} \cup \beta'_1, \{\langle s'_0 \delta \sum_{i=1}^n s'_i x_i, \mu' \rangle\} \cup \beta'_2) =$
 $\text{UNION}(\beta'_1, \beta'_2) \uplus \langle (s_0 \sqcup s'_0) \delta \sum_{i=1}^n (s_i \sqcup s'_i) x_i, \mu \sqcup \mu' \rangle$
if $\langle s_0 \delta \sum_{i=1}^n s_i x_i, \mu \rangle \notin \beta'_1$ and $\langle s'_0 \delta \sum_{i=1}^n s'_i x_i, \mu' \rangle \notin \beta'_2$

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

Example 13 Given the abstract stores $\{\langle 0 = \oplus T, \text{One} \rangle\}$ and $\{\langle \oplus = \oplus T, \text{One} \rangle, \langle 0 < \oplus T, \text{One} \rangle\}$, both $\{\langle 0 = \oplus T, \text{ZeroOrOne} \rangle, \langle \oplus = \oplus T, \text{ZeroOrOne} \rangle, \langle 0 < \oplus T, \text{ZeroOrOne} \rangle\}$ (applying rule 2 thrice) and $\{\langle T = \oplus T, \text{One} \rangle, \langle 0 < \oplus T, \text{ZeroOrOne} \rangle\}$ (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) $\theta \in Cc_i(\beta_1) \vee \theta \in Cc_i(\beta_2) \Rightarrow \theta \in Cc_i(\text{UNION}(\beta_1, \beta_2)).$
- (ii) $\theta \in Cc(\beta_1) \vee \theta \in Cc(\beta_2) \Rightarrow \theta \in Cc(\text{UNION}(\beta_1, \beta_2)).$

3.3.4 Projection

We now turn to the projection operation. Figure 3.1 describes a simple projection algorithm based on the traditional Gaussian and Fourier eliminations which are standard

```

Cproject_set( $\theta, V$ ) {
  if  $V = \emptyset$  return  $\theta$ ;
  else let  $V = \{v\} \sqcup V' \wedge v \notin V'$  in
    return Cproject_set(Cproject( $\theta, v$ ),  $V'$ );
}

Cproject( $\theta, v$ ) {
  if  $\theta = \emptyset$  return  $\emptyset$ ;
  else let  $\theta = \{\lambda\} \sqcup \theta'$  in
    case op( $\lambda$ ) is '='  $\wedge \lambda[v] > 0$  :
      return Cgauss( $\theta', \lambda, v$ );
    case op( $\lambda$ ) is '='  $\wedge \lambda[v] < 0$  :
      return Cgauss( $\theta', \text{Cneg}(\lambda), v$ );
    otherwise:
      return Cfourier( $\theta, v$ );
}

Cgauss( $\theta, \lambda, v$ ) {
  if  $\theta = \emptyset$  return  $\emptyset$ ;
  else let  $\theta = \{\lambda'\} \sqcup \theta'$  in
    return Cgauss( $\theta', \lambda, v$ )  $\sqcup$ 
      {Celiminate( $\lambda', \lambda, v$ )};
}

Cfourier( $\theta, v$ ) {
  if  $\theta = \emptyset$  return  $\emptyset$ ;
  else let  $\theta = \{\lambda\} \sqcup \theta'$  in
     $\langle \theta_1^0, \theta_1^+, \theta_1^- \rangle := \text{Csplit}(\{\lambda\}, v)$ ;
     $\langle \theta_2^0, \theta_2^+, \theta_2^- \rangle := \text{Csplit}(\theta', v)$ ;
    return  $\theta_1^0 \sqcup \text{Cfourier\_step}(\theta_1^+, \theta_2^-, v)$ 
       $\sqcup \text{Cfourier\_step}(\theta_2^+, \theta_1^-, v)$ 
       $\sqcup \text{Cfourier}(\theta', v)$ ;
}

Cfourier_step( $\theta^+, \theta^-, v$ ) {
   $\theta := \emptyset$ ;
  for each  $\lambda^+ \in \theta^+$  and  $\lambda^- \in \theta^-$ 
     $\theta := \theta \sqcup \{\text{Ccombine}(\lambda^-, \lambda^+, v)\}$ ;
  return  $\theta$ ;
}

Csplit( $\theta, v$ ) {
  return Csplit_basic( $\theta, v$ )
}

Cneg( $\lambda$ ) {
  for  $i = 0, \dots, n$ 
     $\lambda'[i] := -\lambda[i]$ ;
  op( $\lambda'$ ) := '=';
  return  $\lambda'$ ;
}

Celiminate( $\lambda_1, \lambda_\pm, v$ ) {
  for  $i := 0, \dots, n$ 
     $\lambda[i] := \lambda_\pm[v] \times \lambda_1[i] - \lambda_1[v] \times \lambda_\pm[i]$ ;
   $\lambda[v] := 0$ ;
  op( $\lambda$ ) := op( $\lambda_1$ );
  return  $\lambda$ ;
}

Ccombine( $\lambda^-, \lambda^+, v$ ) {
  for  $i := 0, \dots, n$ 
     $\lambda[i] := \lambda^+[v] \times \lambda^-[i] - \lambda^-[v] \times \lambda^+[i]$ ;
   $\lambda[v] := 0$ ;
  op( $\lambda$ ) := op( $\lambda^-$ )  $\bowtie$  op( $\lambda^+$ );
  return  $\lambda$ ;
}

```

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

```

Csplit_basic( $\theta, v$ ) {
  if  $\theta = \emptyset$  return  $\langle \emptyset, \emptyset, \emptyset \rangle$ ;
  else let  $\theta = \{\lambda\} \sqcup \theta'$  in
     $\langle \theta^0, \theta^+, \theta^- \rangle := \text{Csplit\_basic}(\theta', v)$ ;
    case  $\lambda[v] = 0$ :
      return  $\langle \theta^0 \sqcup \{\lambda\}, \theta^+, \theta^- \rangle$ ;
    case op( $\lambda$ ) is '='  $\wedge \lambda[v] > 0$ :
      return  $\langle \theta^0, \theta^+ \sqcup \{\lambda\}, \theta^- \sqcup \{\text{Cneg}(\lambda)\} \rangle$ ;
    case op( $\lambda$ ) is '='  $\wedge \lambda[v] < 0$ :
      return  $\langle \theta^0, \theta^+ \sqcup \{\text{Cneg}(\lambda)\}, \theta^- \sqcup \{\lambda\} \rangle$ ;
    case op( $\lambda$ ) is not '='  $\wedge \lambda[v] > 0$ :
      return  $\langle \theta^0, \theta^+ \sqcup \{\lambda\}, \theta^- \rangle$ ;
    case op( $\lambda$ ) is not '='  $\wedge \lambda[v] < 0$ :
      return  $\langle \theta^0, \theta^+, \theta^- \sqcup \{\lambda\} \rangle$ ;
}

```

Figure 3.1: Concrete Projection Algorithms


```

Aproject_set( $\beta, V$ ) {
  if  $V = \emptyset$  return  $\beta$ ;
  else let  $V = \{v\} \cup V' \wedge v \notin V'$  in
    return Aproject_set(Aproject( $\beta, v$ ),  $V'$ );
}

Aproject( $\beta, v$ ) {
  if  $\beta = \emptyset$  return  $\emptyset$ ;
  else let  $\beta = \{\gamma\} \cup \beta' \wedge \gamma = \langle \sigma, \mu \rangle \notin \beta'$  in
    case  $\mu = \text{One} \wedge \text{op}(\sigma)$  is '='  $\wedge \sigma[v] = \oplus$ :
      return Agauss( $\beta', \sigma, v$ );
    case  $\mu = \text{One} \wedge \text{op}(\sigma)$  is '='  $\wedge \sigma[v] = \ominus$ :
      return Agauss( $\beta', \text{Aneg}(\sigma), v$ );
    otherwise:
      return Afourier( $\beta, v$ );
}

Agauss( $\beta, \sigma, v$ ) {
  if  $\beta = \emptyset$  return  $\emptyset$ ;
  else let  $\beta = \{\gamma'\} \cup \beta', \gamma' = \langle \sigma', \mu' \rangle \notin \beta'$  in
    return Agauss( $\beta', \sigma, v$ )  $\uplus$ 
      (Aeliminate( $\sigma', \sigma, v$ ),  $\mu'$ );
}

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

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

Asplit( $\beta, v$ ) {
  return Asplit_basic(Asplit_top( $\beta, v$ ),  $v$ );
}

Aneg( $\sigma$ ) {
  for  $i := 0, \dots, n$ 
     $\sigma'[i] := -\sigma[i]$ ;
   $\text{op}(\sigma') := \text{'='}$ ; return  $\sigma'$ ;
}

Aeliminate( $\sigma_1, \sigma_2^{\pm}, v$ ) {
  for  $i := 0, \dots, n$ 
     $\sigma[i] := \sigma_2^{\pm}[v] \times \sigma_1[i] - \sigma_1[v] \times \sigma_2^{\pm}[i]$ ;
   $\sigma[v] := 0$ ;  $\text{op}(\sigma) := \text{op}(\sigma_1)$ ;
  return  $\sigma$ ;
}

Acombine( $\sigma^-, \sigma^+, v$ ) {
  for  $i := 0, \dots, n$ 
     $\sigma[i] := \sigma^+[v] \times \sigma^-[i] - \sigma^-[v] \times \sigma^+[i]$ ;
   $\sigma[v] := 0$ ;  $\text{op}(\sigma) := \text{op}(\sigma^-) \bowtie \text{op}(\sigma^+)$ ;
  return  $\sigma$ ;
}

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

Asplit_basic( $\beta, v$ ) {
  if  $\beta = \emptyset$  return  $\langle \emptyset, \emptyset, \emptyset \rangle$ ;
  else let  $\beta = \{\gamma\} \cup \beta' \wedge \gamma = \langle \sigma, \mu \rangle \notin \beta'$  in
     $\langle \beta^0, \beta^+, \beta^- \rangle := \text{Asplit\_basic}(\beta', v)$ ;
    case  $\sigma[v] = 0$ :
      return  $\langle \beta^0 \uplus \gamma, \beta^+, \beta^- \rangle$ ;
    case  $\text{op}(\sigma)$  is '='  $\wedge \sigma[v] = \oplus$ :
      return  $\langle \beta^0, \beta^+ \uplus \gamma, \beta^- \uplus \langle \text{Aneg}(\sigma), \mu \rangle \rangle$ ;
    case  $\text{op}(\sigma)$  is '='  $\wedge \sigma[v] = \ominus$ :
      return  $\langle \beta^0, \beta^+ \uplus \langle \text{Aneg}(\sigma), \mu \rangle, \beta^- \uplus \gamma \rangle$ ;
    case  $\text{op}(\sigma)$  is not '='  $\wedge \sigma[v] = \oplus$ :
      return  $\langle \beta^0, \beta^+ \uplus \gamma, \beta^- \rangle$ ;
    case  $\text{op}(\sigma)$  is not '='  $\wedge \sigma[v] = \ominus$ :
      return  $\langle \beta^0, \beta^+, \beta^- \uplus \gamma \rangle$ ;
}

```

Figure 3.2: Abstract Projection Algorithms

in this area. Figure 3.2 presents the abstract algorithm. The algorithms are close to those in [37] but they are simplified thanks to the introduction of operations `Csplit` and `Asplit` which avoids much of the tedious case analysis. The algorithms are also more precise.

The intuition behind the concrete version is as follows. `Cproject` nondeterministically chooses a constraint in the store. If the constraint is an equation whose coefficient for x_v is non-zero, Gaussian elimination is performed. Otherwise, Fourier elimination is used. Gaussian elimination uses the equation or its negation to eliminate x_v from each of the other constraints in the store. The elimination is achieved by applying `Celiminate` on a pair of constraints. Fourier elimination considers each constraint in turn, partitions the store once again, and uses Fourier elimination on the compatible pairs. The presentation is slightly more detailed 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 \mathbb{R} . More formally,

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

The correctness of the abstract version can be proven by showing the consistency of the various operations. Operations `Aneg`, `Aeliminate`, and `Acombine` mimic `Cneg`, `Celiminate`, and `Ccombine` respectively, by performing operations on signs instead of on coefficients. They satisfy the following consistency conditions, which follow directly from the consistency of the operations on signs.

Lemma 6 [Negate] Let σ be an abstract constraint such that $\text{op}(\sigma)$ is '=' and λ be a constraint.

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

Lemma 7 [Eliminate] Let σ_1, σ_{\pm}^+ be abstract constraints such that $\sigma_{\pm}^+[v] = \oplus \wedge$

$op(\sigma_{\pm}^+)$ is '=' , $\lambda_1, \lambda_{\pm}^+$ be constraints and $v \in \mathbb{N}$.

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

Lemma 8 [Combine] Let σ^-, σ^+ be abstract constraints such that $\sigma^+[v] = \oplus \wedge \sigma^-[v] = \ominus$, λ^-, λ^+ be constraints and $v \in \mathbb{N}$.

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

Operation **Agauss** mimics operation **Cgauss** line by line. Contrary to the algorithm in [37], the abstract Gaussian elimination algorithm does not split the \top coefficients of the variable being eliminated into 0, \oplus and \ominus . As shown in Section 3.8, this enables it to be more precise in some cases. Its consistency condition is as follows.

Lemma 9 [Gauss] Let $v \in \mathbb{N}$, σ be an abstract constraint such that $\sigma[v] = \oplus$ and $op(\sigma)$ is '=' and β be an abstract store. We have

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

Example 14 Consider the mortgage example **mg/4** when called with the top level input pattern $\{\langle \top = \oplus P, \text{One} \rangle, \langle \top = \oplus R, \text{One} \rangle\}$. 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

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

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

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

The resulting abstract store is

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

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

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

The resulting abstract store (which represents the input to the recursive call to `mg`) is

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

Note that the equation $\top = \oplus P$ 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 . The multiplicity of the abstract constraints is adjusted so that `Asplit_top` satisfies the following consistency condition.

Lemma 10 [Split Top] Let θ be a constraint store, let β be an abstract store, and $v \in \aleph$. We have

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

`CSplit_basic` and `ASplit_basic` split a (concrete or abstract) store into three partitions with the coefficient of x_v positive, negative or zero.

Lemma 11 [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

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

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. Operation `Csplit` satisfies the following consistency condition.

Lemma 12 [Split] Let θ be a constraint store, let β be an abstract store, and $v \in \aleph$. We have

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

We now come to the operation `Cfourier_step` and its abstract counterpart `Afourier_step`.

Lemma 13 [Fourier Step] Let $v \in \mathbb{N}$, let β^+ be an abstract store such that for all $\gamma \in \beta^+$, $\gamma = \langle \sigma, \mu \rangle$ and $\sigma[v] = \oplus$, let β^- be an abstract store such that for all $\gamma \in \beta^-$, $\gamma = \langle \sigma, \mu \rangle$ and $\sigma[v] = \ominus$. We have

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

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 [37]. This is achieved by testing the multiplicity of the constraint.

Lemma 14 [Fourier] Let $v \in \mathbb{N}$, θ be a store, and β be an abstract store. We have

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

Example 15 Let $\gamma_1 = \langle 0 \leq \oplus R + \oplus B, \text{One} \rangle$, $\gamma_2 = \langle 0 = \oplus P + \top R + \ominus B, \text{One} \rangle$ and $\beta = \{\gamma_1, \gamma_2\}$. Then

$$\begin{aligned} \text{Asplit}(\{\gamma_1\}, R) &= \langle \beta_1^0, \beta_1^+, \beta_1^- \rangle = \langle \emptyset, \{ \langle 0 \leq \oplus R + \oplus B, \text{One} \rangle \}, \emptyset \rangle \\ \text{Asplit}(\{\gamma_2\}, R) &= \langle \beta_2^0, \beta_2^+, \beta_2^- \rangle = \langle \{ \langle 0 = \oplus P + \ominus B, \text{ZeroOrOne} \rangle \}, \\ &\quad \{ \langle 0 = \oplus P + \oplus R + \ominus B, \text{ZeroOrOne} \rangle \}, \\ &\quad \{ \langle 0 = \oplus P + \ominus R + \ominus B, \text{ZeroOrOne} \rangle \} \rangle \end{aligned}$$

According to the algorithm,

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

By similar reasoning, we have that

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

This gives

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

We now state the main result of this section: the consistency of projection.

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

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

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. They satisfy a similar consistency condition, obtained by repeated application of Theorem 7.

Corollary 1 [Project Set] Let $V \in 2^{\mathbb{N}}$, θ be a store and β be an abstract store.

$$\theta \in Cc(\beta) \Rightarrow Cproject_set(\theta, V) \in Cc(Aproject_set(\beta, V)).$$

3.4 Applications

We now present the application of the domain `LSign` to answer questions about satisfiability and unsatisfiability of constraint stores, conditional satisfiability of constraint stores, redundancy of constraints, and freeness of variables.

3.4.1 Satisfiability of Constraint Stores

The first application of the `LSign` domain that we consider is the problem of determining whether a constraint store is satisfiable. In the concrete domain, the function `Cis_sat` : $CS_D \rightarrow \text{Boolean}$ takes a constraint store θ and returns a Boolean value which is true if θ is satisfiable, and false otherwise. The definition of `Cis_sat` is given in Figure 3.3. 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 3.3, makes a conservative approximation to `Cis_sat` that satisfies the following specification.

Specification 15 `Ais_sat` : $AS_D \rightarrow \text{Boolean}$ should satisfy the following consistency condition. $\forall \theta \in CS_D, \forall \beta \in AS_D$:

$$\theta \in Cc(\beta) \Rightarrow (Ais_sat(\beta) \Rightarrow Cis_sat(\theta))$$

<pre> Cis_sat(θ) { $\theta_p = \text{Cproject_set}(\theta, \aleph)$; return Cis_triv_sat($\theta_p$); } Cis_triv_sat($\theta$) { if $\theta = \emptyset$ return true ; else let $\theta = \{\lambda\} \sqcup \theta'$ in case λ is $c_0 < \sum_{i=1}^n 0x_i \wedge c_0 < 0$: case λ is $c_0 = \sum_{i=1}^n 0x_i \wedge c_0 = 0$: case λ is $c_0 \leq \sum_{i=1}^n 0x_i \wedge c_0 \leq 0$: return Cis_triv_sat(θ'); otherwise: return false; } </pre>	<pre> Ais_sat(β) { $\beta_p = \text{Aproject_set}(\beta, \aleph)$; return Ais_triv_sat($\beta_p$); } Ais_triv_sat($\beta$) { if $\beta = \emptyset$ return true ; else let $\beta = \{\gamma\} \cup \beta' \wedge \gamma = \langle \sigma, \mu \rangle \notin \beta'$ in case σ is $s_0 < \sum_{i=1}^n 0x_i \wedge s_0 = \ominus$: case σ is $s_0 = \sum_{i=1}^n 0x_i \wedge s_0 = 0$: case σ is $s_0 \leq \sum_{i=1}^n 0x_i \wedge (s_0 = 0 \vee s_0 = \ominus)$: return Ais_triv_sat(β'); otherwise: return false; } </pre>
--	--

Figure 3.3: Satisfiability Algorithms

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

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

Example 18 Projecting all the variables from $\{\langle \top \leq \oplus R + \oplus B, \text{One} \rangle, \langle 0 = \oplus B, \text{One} \rangle, \langle \oplus = \oplus R, \text{One} \rangle\}$ gives the abstract store $\{\langle \top \leq 0, \text{One} \rangle\}$, 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.

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

3.4.2 Unsatisfiability of Constraint Stores

The second application of the `LSign` domain that we consider is the problem of determining whether a constraint store is unsatisfiable. In the concrete domain, the function $\text{Cis_unsat} : CS_D \rightarrow \text{Boolean}$ takes a constraint store θ and returns a Boolean value

```

Cis_unsat( $\theta$ ) {
 $\theta_p = \text{Cproject\_set}(\theta, \aleph)$ ;
return Cis_triv_unsat( $\theta_p$ );
}

Cis_triv_unsat( $\theta$ ) {
if  $\theta = \emptyset$  return true ;
else let  $\theta = \{\lambda\} \sqcup \theta'$  in
  case  $\lambda$  is  $c_0 < \sum_{i=1}^n 0x_i \wedge c_0 \geq 0$ :
  case  $\lambda$  is  $c_0 = \sum_{i=1}^n 0x_i \wedge c_0 \neq 0$ :
  case  $\lambda$  is  $c_0 \leq \sum_{i=1}^n 0x_i \wedge c_0 > 0$ :
    return true;
  otherwise:
    return Cis_triv_unsat( $\theta'$ );
}

Ais_unsat( $\beta$ ) {
 $\beta_p = \text{Aproject\_set}(\beta, \aleph)$ ;
return Ais_triv_unsat( $\beta_p$ );
}

Ais_triv_unsat( $\beta$ ) {
if  $\beta = \emptyset$  return true ;
else let  $\beta = \{\gamma\} \cup \beta' \wedge \gamma = \langle \sigma, \mu \rangle \notin \beta'$  in
  case  $\sigma$  is  $s_0 < \sum_{i=1}^n 0x_i \wedge (s_0 = 0 \vee s_0 = \oplus) \wedge \mu = \text{One}$ :
  case  $\sigma$  is  $s_0 = \sum_{i=1}^n 0x_i \wedge (s_0 = \oplus \vee s_0 = \ominus) \wedge \mu = \text{One}$ :
  case  $\sigma$  is  $s_0 \leq \sum_{i=1}^n 0x_i \wedge s_0 = \oplus \wedge \mu = \text{One}$ :
    return true;
  otherwise:
    return Ais_triv_unsat( $\beta'$ );
}

```

Figure 3.4: Unsatisfiability Algorithms

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 3.4. 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 3.4, makes a conservative approximation to `Cis_unsat` that satisfies the following specification.

Specification 16 $\text{Ais_unsat} : AS_D \rightarrow \text{Boolean}$ should satisfy the following consistency condition. $\forall \theta \in CS_D, \forall \beta \in AS_D$:

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

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

Example 19 Projecting all the variables from $\{0 \leq R + B, 0 = B, -3 = R\}$ gives the constraint store $\{3 \leq 0\}$, indicating that the original constraint store is unsatisfiable.

Example 20 Projecting all the variables from $\{\langle 0 \leq \oplus R + \oplus B, \text{One} \rangle, \langle 0 = \oplus B, \text{One} \rangle, \langle \ominus = \oplus R, \text{One} \rangle\}$ gives the abstract store $\{\langle \oplus \leq 0, \text{One} \rangle\}$, indicating that all the

constraint stores in its concretization are unsatisfiable.

Example 21 Projecting all the variables from $\{\langle \top \leq \oplus R + \oplus B, \text{One} \rangle, \langle 0 = \oplus B, \text{One} \rangle, \langle \oplus = \oplus R, \text{One} \rangle\}$ gives the abstract store $\{\langle \top \leq 0, \text{One} \rangle\}$, 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.

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

3.4.3 Conditional Satisfiability of Constraint Stores

The next application of `LSign` that we consider is the problem of conditional satisfiability of constraint stores. In the concrete domain, the function `Cis_cond_sat` : $CS_D \times CS_D \rightarrow \text{Boolean}$ takes two constraint stores θ_1 and θ_2 and returns true if the conjunction $\theta_1 \sqcup \theta_2$ is satisfiable whenever θ_1 is satisfiable. In other words,

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

Its abstract counterpart `Ais_cond_sat` is specified as follows.

Specification 17 `Ais_cond_sat` : $AS_D \times AS_D \rightarrow \text{Boolean}$ should satisfy the following consistency condition. $\forall \theta_1, \theta_2 \in CS_D, \forall \beta_1, \beta_2 \in AS_D$:

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

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 $\text{CLP}(\mathcal{R}_{Lin})$ 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

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

The correctness of this implementation w.r.t. the specification is an easy consequence of the correctness of `Ais_sat`, `Ais_unsat` and the \sqcup 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 $\theta_1 \sqcup \theta_2$ 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 $\theta_1 \sqcup \theta_2$. 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 $\beta_1 \sqcup \beta_2$ only for the stores in the concretization of β_1 that are satisfiable. The approach taken is to remove as many unsatisfiable stores as possible from the concretization of β_1 , before computing `Ais_sat`($\beta_1 \sqcup \beta_2$). Intuitively, this approach leads to a more precise approximation of `Cis_cond_sat`(θ_1, θ_2) because we only need to check the satisfiability of $\theta_1 \sqcup \theta_2$ 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 3.5) 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 so it is an equivalence transformation for satisfiable constraint stores, i.e., the following proposition holds.

Proposition 1 Let θ be a constraint store. Then

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

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 3.5) is important to improve the accuracy of conditional satisfiability in the abstract domain.

Example 22 Let $\beta_1 = \{\langle T = \oplus X, \text{One} \rangle, \langle T < 0, \text{One} \rangle\}$ and $\beta_2 = \{\langle 0 = \oplus X + \oplus Y, \text{One} \rangle\}$. 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 $\text{Cis_sat}(\theta_1 \sqcup \theta_2)$ is true. We expect therefore that $\text{Ais_cond_sat}(\beta_1, \beta_2)$ returns true. However we see that

$$\begin{aligned} \text{Ais_cond_sat}(\beta_1, \beta_2) &= \text{Ais_unsat}(\beta_1) \vee \text{Ais_sat}(\beta_1 \uplus \beta_2) \\ &= \text{false} \vee \text{Ais_triv_sat}(\{\langle \top < 0, \text{One} \rangle\}) \\ &= \text{false}. \end{aligned}$$

This inaccuracy arises because of the possibly unsatisfiable ground constraint $\langle \top < 0, \text{One} \rangle$ in β_1 . It can be observed however that there is another abstract store $\beta'_1 = \{\langle \top = \oplus X, \text{One} \rangle\}$ which includes all the satisfiable constraint stores of $Cc(\beta_1)$ in its concretization and such that $\text{Ais_sat}(\beta'_1 \uplus \beta_2)$ is true. Moreover, $\beta'_1 = \text{Ared_gnd}(\beta_1)$.

An improved implementation of Ais_cond_sat , may therefore be given by

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

Eliminating Equations In the concrete domain, the transformation Cred_eqn (Figure 3.5) looks for an equation for each variable, and if possible, uses the equation to Gauss 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 $v \in \mathbb{N}$, $V \in 2^{\mathbb{N}}$ and θ be a store. Then

$$\theta \leftrightarrow \text{Cred_eqn_step}(\theta, v) \leftrightarrow \text{Cred_eqn_set}(\theta, V) \leftrightarrow \text{Cred_eqn}(\theta).$$

The following example indicates why the abstraction of Cred_eqn , i.e. Ared_eqn (given in Figure 3.5) is important to improve the accuracy of conditional satisfiability in the abstract domain.

Example 23 Let $\beta_1 = \{\langle \oplus = \oplus X, \text{One} \rangle, \langle \top < \oplus X, \text{One} \rangle\}$ and $\beta_2 = \{\langle 0 = \oplus X + \oplus Y, \text{One} \rangle\}$. 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 $\text{Cis_sat}(\theta_1 \sqcup \theta_2)$ is true. We expect therefore

that $\text{Ais_cond_sat}(\beta_1, \beta_2)$ returns true. However we see that

$$\begin{aligned}\text{Ais_cond_sat}(\beta_1, \beta_2) &= \text{Ais_unsat}(\beta_1) \vee \text{Ais_sat}(\beta_1 \uplus \beta_2) \\ &= \text{false} \vee \text{Ais_triv_sat}(\{\langle \top < 0, \text{One} \rangle\}) \\ &= \text{false}.\end{aligned}$$

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 $\beta'_1 = \{\langle \oplus = \oplus X, \text{One} \rangle\}$ which includes all the satisfiable constraint stores of $Cc(\beta_1)$ in its concretization (under equivalence closure) and such that $\text{Ais_sat}(\beta'_1 \uplus \beta_2)$ is true. Moreover, $\beta'_1 = \text{Ared_gnd}(\text{Ared_eqn}(\beta_1))$.

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

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

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 24 Let $\beta_1 = \{\langle \top = \oplus X, \text{One} \rangle, \langle 0 < \oplus X, \text{One} \rangle\}$ and $\beta_2 = \{\langle 0 = \oplus X + \oplus Y, \text{One} \rangle\}$. 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 $\text{Cis_sat}(\theta_1 \sqcup \theta_2)$ is true. We expect therefore that $\text{Ais_cond_sat}(\beta_1, \beta_2)$ returns true. However we see that

$$\begin{aligned}\text{Ais_cond_sat}(\beta_1, \beta_2) &= \text{Ais_unsat}(\beta_1) \vee \text{Ais_sat}(\beta_1 \uplus \beta_2) \\ &= \text{false} \vee \text{Ais_triv_sat}(\{\langle \top < 0, \text{One} \rangle\}) \\ &= \text{false}.\end{aligned}$$

This inaccuracy arises because of the top coefficient in β_1 . It can be observed that there is another abstract store $\beta'_1 = \{\langle \oplus = \oplus X, \text{One} \rangle, \langle 0 < \oplus X, \text{One} \rangle\}$ which includes all the satisfiable constraint stores of $Cc(\beta_1)$ in its concretization and such that $\text{Ais_sat}(\beta'_1 \uplus \beta_2)$ 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 3.5) 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 3.5). In the concrete domain, the operation $\text{Creduce} : CS_D \rightarrow CS_D$ consists of the composition of `Cred_eqn` and `Cred_gnd`. In the abstract domain, the operation $\text{Areduce} : AS_D \rightarrow AS_D$ 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 $\text{Areduce}(\beta)$ contain all the satisfiable stores from the concretization of β , it does not preclude the introduction of stores into $Cc(\text{Areduce}(\beta))$ 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

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

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

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

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

Example 25 Let $\beta_1 = \{\langle \top < \top X, \text{One} \rangle\}$ and $\beta_2 = \{\langle 0 = \oplus X + \oplus Y, \text{One} \rangle\}$. 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 $\text{Cis_sat}(\theta_1 \sqcup \theta_2)$. We expect therefore that $\text{Ais_cond_sat}(\beta_1, \beta_2)$ returns

```

Creduce( $\theta$ ) {
  return Cred_gnd(Cred_eqn( $\theta$ ));
}

Cred_gnd( $\theta$ ) {
  if  $\theta = \emptyset$  return  $\emptyset$ ;
  else let  $\theta = \{\lambda\} \sqcup \theta'$  in
    case  $\lambda$  is  $c_0 \delta \sum_{i=1}^n 0x_i$ :
      return Cred_gnd( $\theta'$ );
    otherwise:
      return Cred_gnd( $\theta'$ )  $\sqcup \{\lambda\}$ ;
}

Cred_eqn( $\theta$ ) {
  return Cred_eqn_set( $\theta, \mathbb{N}$ );
}

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

Cred_eqn_step( $\theta, v$ ) {
  if  $\theta = \emptyset$  return  $\emptyset$ ;
  else let  $\theta = \{\lambda\} \sqcup \theta'$  in
    case  $\text{op}(\lambda)$  is '='  $\wedge \lambda[v] > 0$ :
      return Cgauss( $\theta', \lambda, v$ )  $\sqcup \{\lambda\}$ ;
    case  $\text{op}(\lambda)$  is '='  $\wedge \lambda[v] < 0$ :
      return Cgauss( $\theta', \text{Cneg}(\lambda), v$ )  $\sqcup \{\lambda\}$ ;
    otherwise:
      return  $\theta$ ;
}

Areduce( $\beta$ ) {
  return Ared_gnd(Ared_eqn(Ared_top( $\beta$ )));
}

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

Ared_eqn( $\beta$ ) {
  return Ared_eqn_set( $\beta, \mathbb{N}$ );
}

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

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

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

```

Figure 3.5: Reduction Algorithms

true. However we see that

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

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

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

$$\begin{aligned}
&\text{Cis_sat}(\theta_1 \sqcup \theta_2) \\
&= \text{Cis_triv_sat}(\text{Cproject_set}(\theta_1 \sqcup \theta_2, \aleph)) \\
&= \text{Cis_triv_sat}(\text{Cproject_set}(\text{Cproject_set}(\theta_1, \text{Var}(\theta_1) \setminus \text{Var}(\theta_2)) \sqcup \theta_2, \aleph)) \\
&= \text{Cis_sat}(\text{Cproject_set}(\theta_1, \text{Var}(\theta_1) \setminus \text{Var}(\theta_2)) \sqcup \theta_2)
\end{aligned}$$

This suggests the following implementation for conditional satisfiability.

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

$$\begin{aligned}
&\text{Ais_cond_sat}(\beta_1, \beta_2) \\
&= \text{Ais_unsat}(\beta_1) \vee \text{Ais_sat}(\text{Areduce}(\text{Aproject_set}(\beta_1, \text{Var}(\beta_1) \setminus \text{Var}(\beta_2))) \uplus \beta_2).
\end{aligned}$$

where $\text{Var}(\beta)$ 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

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

Summary The naive implementation of $\text{Ais_cond_sat}(\beta_1, \beta_2)$ 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.

3.4.4 Redundancy of Constraints

The next application of the LSign domain is the problem of determining whether a given constraint is redundant in the context of a given constraint store. It is convenient to generalize this problem to the problem of determining whether a constraint store is redundant in the context of another given constraint store⁵. In the concrete domain, the function `Cis_redundant` : $CS_D \times CS_D \rightarrow \text{Boolean}$ takes a constraint store θ_r and returns true if it can be determined to be redundant in the context of the constraint store θ . The function is conservative even in the concrete domain in the following manner:

$$\text{Cis_redundant}(\theta_r, \theta) \Rightarrow \theta_r \text{ is redundant in the context of } \theta$$

`Cis_redundant` performs the following conservative check for redundancy. First it expresses θ on the variables of θ_r by eliminating the variables not in θ_r . It then applies `Creduce` on the resulting store in order to make the information available on the subsystem consisting of the variables of θ_r more explicit. The next step is to call `Csimplify` which simplifies θ_r using the equations of θ . For each variable v , it checks whether v occurs with a nonzero coefficient in an equation in the store θ and if such an equation is found, it can be used to substitute the value of v into θ_r by Gauss elimination. The final step is to check if the resulting simplified store is trivially satisfiable. The algorithm for `Cis_redundant` is given in Figure 3.6. Its abstract counterpart `Ais_redundant` is also defined in Figure 3.6 and can be specified as follows.

⁵By definition, θ_r is redundant in the context of θ if $\theta \leftrightarrow \theta \uplus \theta_r$.

<pre> Cis_redundant(θ_r, θ) { $\theta_p := \text{Creduce}(\text{Cproject_set}(\theta, \mathbb{N} \setminus \text{Var}(\theta_r)))$; $\theta_s := \text{Csimplify}(\theta_r, \theta_p, \mathbb{N})$; return Cis_triv_sat($\theta_s$); } Csimplify($\theta_r, \theta, V$) { if $V = \emptyset$ return θ_r; else let $V = \{v\} \cup V' \wedge v \notin V'$ in $\theta_v := \text{Csimplify_step}(\theta_r, \theta, v)$; return Csimpify($\theta_v, V'$); } Csimplify_step($\theta_r, \theta, v$) { if $\theta = \emptyset$ return θ_r; else let $\theta_v = \{\lambda\} \sqcup \theta'$ in case op(λ) is '=' $\wedge \lambda[v] > 0$: return Cgauss(θ_r, λ, v); case op(λ) is '=' $\wedge \lambda[v] < 0$: return Cgauss($\theta_r, \text{Cneg}(\lambda), v$); otherwise: return θ_r; } </pre>	<pre> Ais_redundant(β_r, β) { $\beta_p := \text{Areduce}(\text{Aproject_set}(\beta, \mathbb{N} \setminus \text{Var}(\beta_r)))$; $\beta_s = \text{Asimplify}(\beta_r, \beta_p, \mathbb{N})$; return Ais_triv_sat($\beta_s$); } Asimplify($\beta_r, \beta, V$) { if $V = \emptyset$ return β_r; else let $V = \{v\} \cup V' \wedge v \notin V'$ in $\beta_v := \text{Asimplify_step}(\beta_r, \beta, v)$; return Asimplify($\beta_v, V'$); } Asimplify_step($\beta_r, \beta, v$) { if $\beta = \emptyset$ return β_r; else let $\beta_v = \{\gamma\} \cup \beta' \wedge \gamma = \langle \sigma, \mu \rangle \notin \beta'$ in case $\mu = \text{One} \wedge \text{op}(\sigma)$ is '=' $\wedge \sigma[v] = \oplus$: return Agauss($\beta_r, \sigma, v$); case $\mu = \text{One} \wedge \text{op}(\sigma)$ is '=' $\wedge \sigma[v] = \ominus$: return Agauss($\beta_r, \text{Aneg}(\sigma), v$); otherwise: return β_r; } </pre>
---	--

Figure 3.6: Redundancy Algorithms

Specification 18 $\text{Ais_redundant} : AS_D \times AS_D \rightarrow \text{Boolean}$ should satisfy the following consistency condition. $\forall \theta \in CS_D, \forall \beta_r \in AS_D, \forall \beta \in AS_D$:

$$\begin{aligned} & \theta_r \in Cc(\beta_r) \wedge \theta \in Cc(\beta) \wedge \theta \text{ satisfiable} \\ & \Rightarrow (\text{Ais_redundant}(\beta_r, \beta) \Rightarrow \text{Cis_redundant}(\theta_r, \theta)). \end{aligned}$$

Example 26 Let β be the abstract store

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

and β_r be the abstract store

$\{\langle 0 < \oplus T, \text{One} \rangle\}.$

$\text{Areduce}(\text{Aproject_set}(\beta, \{P, R, B, P1, T1\}))$ is

$\{\langle \oplus = \oplus T, \text{One} \rangle\}.$

<pre> Cis_free(θ, v) { if $\theta = \emptyset$ return true; else let $\theta = \{\lambda\} \sqcup \theta'$ in case $\lambda[v] = 0$: return Cis_free(θ', v); otherwise: return false; } </pre>	<pre> Ais_free(β, v) { if $\beta = \emptyset$ return true; else let $\beta = \{\gamma\} \cup \beta' \wedge \gamma = \langle \sigma, \mu \rangle \notin \beta'$ in case $\sigma[v] = 0$: return Ais_free(β', v); otherwise: return false; } </pre>
---	--

Figure 3.7: Freeness Algorithms

The simplified value of β_r is therefore just $\{\langle \ominus < 0, \text{One} \rangle\}$ which is seen to be trivially satisfiable. So $\text{Ais_redundant}(\beta_r, \beta)$ is true. Hence for any $\theta \in Cc(\beta)$ and $\theta_r \in Cc(\beta_r)$, we have that $\text{Cis_redundant}(\theta_r, \theta)$.

The following theorem proves that Ais_redundant is a conservative approximation of Cis_redundant .

Theorem 12 [Is Redundant] Let θ_r, θ be stores and β_r, β be abstract stores.

$$\begin{aligned}
& \theta_r \in Cc(\beta_r) \wedge \theta \in Cc(\beta) \wedge \theta \text{ satisfiable} \\
& \Rightarrow (\text{Ais_redundant}(\beta_r, \beta) \Rightarrow \text{Cis_redundant}(\theta_r, \theta)).
\end{aligned}$$

3.4.5 Freeness of Variables

The final application of the LSign domain that we consider, is the problem of determining whether a given variable is “free” i.e. it does not occur in a given constraint store with a non-zero coefficient. In the concrete domain, the function $\text{Cis_free} : CS_D \times \mathbb{N} \rightarrow \text{Boolean}$ takes a constraint store θ and a variable v and returns a Boolean value which is true if the variable does not occur in the constraint store and false otherwise. The definition of Cis_free is very straightforward and is given in Figure 3.7. Its abstract counterpart Ais_free , also given in Figure 3.7, makes a conservative approximation to Cis_free and can be specified as follows.

Specification 19 $\text{Ais_free} : AS_D \times \mathbb{N} \rightarrow \text{Boolean}$ should satisfy the following consistency condition. $\forall \theta \in CS_D, \forall \beta \in AS_D, \forall v \in \mathbb{N} :$

$$\theta \in Cc(\beta) \wedge \theta \text{ satisfiable} \Rightarrow (\text{Ais_free}(\beta, v) \Rightarrow \text{Cis_free}(\theta, v))$$

Example 27 $\text{Ais_free}(\{\langle T = \oplus P, \text{One} \rangle, \langle T = \oplus R, \text{One} \rangle, \langle 0 \leq \oplus P, \text{One} \rangle\}, P1)$ is true, indicating that when `mg/4` is called with the top level input pattern $\{\langle T = \oplus P, \text{One} \rangle, \langle T = \oplus R, \text{One} \rangle\}$, for the first execution of the second clause, the variable `P1` is free at the program point just after the inequality $P \geq 0$ and just before the equation $P1 = P * 1.01 - R$.

It can be proved that the definition of `Ais_free` satisfies its specification, giving rise to the following theorem.

Theorem 13 [Is Free] Let $v \in \mathbb{N}$, θ be a constraint store and β be an abstract store.

- (i) $\theta \in Cc_i(\beta) \Rightarrow (\text{Ais_free}(\beta, v) \Rightarrow \text{Cis_free}(\theta, v))$.
- (ii) $\theta \in Cc(\beta) \wedge \theta \text{ satisfiable} \Rightarrow (\text{Ais_free}(\beta, v) \Rightarrow \text{Cis_free}(\theta, v))$.

3.5 The Power Domain 2^{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^{LSign} in order to get the required accuracy. This is a fairly standard construction in abstract interpretation [15]. We briefly introduce the domain 2^{LSign} and its abstract operations.

Definition 19 [Abstract Multistoires] An *abstract multistore* over D is a set α of abstract stores. The set of multistoires 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 multistoires are denoted by the letter α , possibly subscripted.

Example 28 Constraint stores such as $\{0 = T\}$, $\{2 = 3T, 0 < T\}$ and $\{3 = T, 0 < 2T\}$ all belong to the concretization of the abstract multistore $\{\{\langle 0 = \oplus T, \text{One} \rangle\}, \{\langle \oplus = \oplus T, \text{One} \rangle, \langle 0 < \oplus T, \text{One} \rangle\}\}$

$$\begin{aligned}
\alpha \uplus \gamma &= \bigcup_{\beta \in \alpha} \beta \uplus \gamma \\
\alpha \uplus \alpha' &= \bigcup_{\beta \in \alpha, \beta' \in \alpha'} \beta \uplus \beta' \\
\text{Aproject}(\alpha, v) &= \bigcup_{\beta \in \alpha} \text{Aproject}(\beta, v) \\
\text{Aproject_set}(\alpha, V) &= \bigcup_{\beta \in \alpha} \text{Aproject_set}(\beta, V) \\
\text{Ais_sat}(\alpha) &= \bigwedge_{\beta \in \alpha} \text{Ais_sat}(\beta) \\
\text{Ais_unsat}(\alpha) &= \bigwedge_{\beta \in \alpha} \text{Ais_unsat}(\beta) \\
\text{Ais_cond_sat}(\alpha_1, \alpha_2) &= \bigwedge_{\beta_1 \in \alpha_1, \beta_2 \in \alpha_2} \text{Ais_cond_sat}(\beta_1, \beta_2) \\
\text{Ais_redundant}(\alpha_r, \alpha) &= \bigwedge_{\beta_r \in \alpha_r, \beta \in \alpha} \text{Ais_redundant}(\beta_r, \beta) \\
\text{Ais_free}(\alpha, v) &= \bigwedge_{\beta \in \alpha} \text{Ais_free}(\beta, v)
\end{aligned}$$

Figure 3.8: Abstract Operations and Applications for 2^{LSign}

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 $\sqsubseteq: \text{AMS}_D \times \text{AMS}_D$ is a pre-order and is monotonic w.r.t. the concretization function. Figure 3.8 contains the definitions of various algorithms for 2^{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^{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.

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

Example 29 The abstract multistore $\{\{\langle 0 = \oplus T, \text{One} \rangle\}, \{\langle \oplus = \oplus T, \text{One} \rangle, \langle 0 < \oplus T, \text{One} \rangle\}\}$ is the upper bound of the abstract multistores $\{\{\langle \oplus = \oplus T, \text{One} \rangle, \langle 0 < \oplus T, \text{One} \rangle\}\}$ and $\{\{\langle 0 = \oplus T, \text{One} \rangle\}\}$.

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

Theorem 14 Let α_1 and α_2 be abstract multistores and θ be a constraint store.

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

Note that it may be possible to avoid the coefficient \top altogether in the power domain 2^{LSign} , thus removing a large source of imprecision. This is because for any \top coefficient

in an abstract store, it can be split into three abstract stores. However, this may cause an explosion in the number of abstract stores that compose a typical abstract multistore. This remains an open area of investigation.

We would also like to mention here that 2^{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

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

3.6 Relation to Abstract Interpretation Framework

As mentioned in Chapter 2, any abstract domain that is used in our abstract interpretation framework needs to provide definitions of operations such as **RESTRC**, **EXTG** etc. We now indicate how these operations can be defined in terms of the operations for 2^{LSign} .

UNION (Upper bound) Operation **UNION** is used to compute the result of a predicate, given the result of its individual clauses. Its abstract version is just the upper bound for abstract multistores.

$$\text{UNION}(\alpha_1, \dots, \alpha_n) = \begin{cases} \alpha_1 & \text{if } n = 1 \\ \text{UNION}(\dots \text{UNION}(\alpha_1, \alpha_2) \dots, \alpha_n) & \text{otherwise} \end{cases}$$

AI_ADD (Addition of a constraint) Operation **AI_ADD** is used when a constraint is encountered in the program. Its abstract version adds the abstraction of the constraint to the input abstract substitution. Let λ be the concrete constraint $c_0 \delta \sum_{i=1}^n c_i x_i$ in the program. Then

$$\text{AI_ADD}(\lambda, \alpha) = \alpha \uplus \langle s_0 \delta \sum_{i=1}^n s_i x_i, \text{One} \rangle \quad \text{where } c_i \in Cc(s_i) \quad (0 \leq i \leq n)$$

RESTRC (Restriction of a clause substitution) Operation **RESTRC** is used at the end of a clause execution to restrict the substitution expressed on all the clause variables to a substitution expressed on only the head variables of the clause. Its abstract version

projects out the variables that occur in the clause body but not in the clause head. Let x_{m+1}, \dots, x_n be the variables that occur only in the body of c . Then

$$\text{RESTRC}(c, \alpha) = \text{Aproject_set}(\alpha, \{x_{m+1}, \dots, x_n\})$$

EXTC (Extension of a clause substitution) Operation EXTC is used to is used at the beginning of a clause's execution to express the input substitution on all the clause variables and not just the head variables. Its abstract version for a clause c is a trivial operation.

$$\text{EXTC}(c, \alpha) = \alpha$$

RESTRG (Restriction of a substitution before a literal) Operation RESTRG is used before the execution of a literal in the body of a clause to compute the input substitution for the execution of the literal. Its abstract version first projects out the variables of the clause that do not appear in the literal from the abstract substitution, and then normalizes the resulting abstract substitution. Let x_{i_1}, \dots, x_{i_m} be the variables that occur in the literal l and let x_1, \dots, x_n be the variables of the clause in which the literal occurs. Then

$$\text{RESTRG}(l, \alpha) = \text{norm } [x_{i_1}, \dots, x_{i_m}] \text{Aproject_set}(\alpha, \{x_1, \dots, x_n\} \setminus \{x_{i_1}, \dots, x_{i_m}\})$$

EXTG (Extension of a substitution after a literal) Operation EXTG is used after the execution of a literal to extend its result to the clause substitution. Its abstract version first denormalizes the abstract substitution α' for the output of the literal l and then adds it to the clause abstract substitution α . Let x_{i_1}, \dots, x_{i_m} be the variables that appear in the literal l , in that order. Then

$$\text{EXTG}(l, \alpha, \alpha') = \alpha \uplus \text{denorm } [x_{i_1}, \dots, x_{i_m}] \alpha'$$

3.7 Complete Example

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

Example 30 [Computation of LSign Output Description for mg/4] The abstract multistore describing the output of the first clause is

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

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

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

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

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

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

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

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.

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

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

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

⊔

$$\{\{\langle 0 = \oplus T1, \text{One} \rangle, \langle 0 = \oplus P1 + \oplus B, \text{One} \rangle\}, \{\langle 0 < \oplus T1, \text{One} \rangle, \langle \oplus = \oplus T1, \text{One} \rangle, \langle 0 \leq \oplus P1, \text{One} \rangle, \langle 0 = \oplus P1 + \oplus R + \oplus B, \text{One} \rangle\}\}$$

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

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

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

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

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

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

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

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

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

Example 31 [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

$$\{ \{ \langle T = \oplus P, \text{One} \rangle, \langle T = \oplus R, \text{One} \rangle \} \}$$

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

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

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

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

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

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

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

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

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

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

\sqcup

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

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

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

Taking the union with the old input gives the updated input

```
{
  {⟨T = ⊕P, One⟩, ⟨T = ⊕R, One⟩},
  {⟨⊖ < ⊕T, One⟩, ⟨T = ⊕R, One⟩, ⟨T = ⊕P + ⊕R, One⟩}
  {⟨⊖ < ⊕T, One⟩, ⟨⊖ < ⊕T, Any⟩, ⟨T = ⊕R, One⟩, ⟨T ≤ ⊖R, One⟩, ⟨T = ⊕P + ⊕R, One⟩}
}
```

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

```
{
  {⟨T = ⊕P, One⟩, ⟨T = ⊕R, One⟩},
  {⟨⊖ < ⊕T, One⟩, ⟨T = ⊕R, One⟩, ⟨T = ⊕P + ⊕R, One⟩}
  {⟨⊖ < ⊕T, One⟩, ⟨⊖ < ⊕T, Any⟩, ⟨T = ⊕R, One⟩, ⟨T ≤ ⊖R, One⟩, ⟨T = ⊕P + ⊕R, One⟩}
  {⟨⊖ < ⊕T, One⟩, ⟨⊖ < ⊕T, Any⟩, ⟨T = ⊕R, One⟩,
    ⟨T ≤ ⊖R, One⟩, ⟨T ≤ ⊖R, Any⟩, ⟨T = ⊕P + ⊕R, One⟩}
}
```

which can be simplified to

```
{
  {⟨T = ⊕P, One⟩, ⟨T = ⊕R, One⟩},
  {⟨⊖ < ⊕T, One⟩, ⟨T = ⊕R, One⟩, ⟨T = ⊕P + ⊕R, One⟩}
  {⟨⊖ < ⊕T, One⟩, ⟨⊖ < ⊕T, Any⟩, ⟨T = ⊕R, One⟩,
    ⟨T ≤ ⊖R, One⟩, ⟨T ≤ ⊖R, Any⟩, ⟨T = ⊕P + ⊕R, One⟩}
}
```

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

3.8 Discussion

The LSign domain was introduced by Marriott and Stuckey [37] as an elegant domain for analyzing constraint logic programming languages over linear real constraints. Its key conceptual idea is the abstraction of linear constraints by replacing coefficients by their signs. Unfortunately, the ordering given in [37], i.e.,

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

does not capture the intended meaning, since it would conclude that the set $\beta_1 = \{\oplus = \oplus x_1 + \oplus x_2, \oplus = \oplus x_1 + \oplus x_2\}$ is smaller than $\beta_2 = \{\oplus = \oplus x_1 + \top x_2\}$, where \oplus represents

a strictly positive coefficient, \ominus a strictly negative coefficient, and \top any coefficient. The intended meaning of [37] 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 [37]. 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`.

The first purpose of this chapter⁶ is to reconsider the domain `LSign` and to correct and complete the results of [37]. In this respect, our first contribution is to define an ordering on this domain and to capture the intended meaning of the domain through a monotone concretization function from abstract constraint stores to sets of constraint stores, where a constraint store is a multiset of constraints.⁷ Our second contribution is to show that ordering two abstract constraint stores reduces to a matching problem, which can be solved in polynomial time. Our third contribution is to reconsider the other abstract operations of `LSign`, i.e., projection, upper bound, and addition of a constraint, and to prove their consistency. In particular, we propose a simpler algorithm for abstract projection which factorizes the case analysis in a single procedure and a schema for generating consistent upper bound operations.

The second purpose of this chapter is to present improvements to the projection algorithm. Improving the accuracy of the projection algorithm can be very important in practice. The first improvement is to make abstract Fourier elimination much more precise. In [37], 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 $\{\langle \top = \top x + \oplus y, \text{ZeroOrOne} \rangle\}$ leads to the store $\{\langle \top = \oplus y, \text{ZeroOrOne} \rangle, \langle \top = \oplus y, \text{Any} \rangle\}$ using the algorithm of [37]. 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

⁶A preliminary version of this chapter titled “`LSign` Reordered” appeared in Static Analysis Symposium, Glasgow, 1995.

⁷Note that Marriott and Stuckey use an abstraction function for constraints and an approximation relation for the remaining semantic objects.

projecting y as it potentially contains more than one different assignment of a value to y . Our improved algorithm would lead to the store $\{\langle \top = \oplus y, \text{ZeroOrOne} \rangle\}$, which can be easily seen to be definitely satisfiable. The second improvement is to make abstract Gauss elimination much more precise. Consider the following example. Using the algorithms of [37], projecting x from $\{\langle \oplus = \oplus x, \text{One} \rangle, \langle \top = \top x + \oplus y, \text{One} \rangle\}$ involves considering the cases 0, \oplus and \ominus for the \top coefficient of x in the second constraint, leading to $\{\langle \top = \oplus y, \text{ZeroOrOne} \rangle, \langle \top = \oplus y, \text{Any} \rangle\}$ which may or may not be satisfiable. However, by directly substituting $\langle \oplus = \oplus x, \text{One} \rangle$ into the second constraint, we get a simpler abstract store $\{\langle \top = \oplus y, \text{One} \rangle\}$ that accurately describes the result of projecting x and which can be deduced to be definitely satisfiable. Our abstract Gauss elimination algorithm produces this store by avoiding the imprecise splitting of the coefficient \top .

The third purpose of this chapter is to present formally the applications of the domain LSign for a variety of analyses including satisfiability analysis, unsatisfiability analysis, conditional satisfiability analysis, redundancy analysis and freeness analysis. We give abstract algorithms that answer a variety of *interesting* questions about satisfiability of constraint stores, redundancy of constraints and freeness of variables. While the utilization of LSign for satisfiability analysis had been mentioned earlier [37], there was no detailed presentation of conditional satisfiability. Also, redundancy and freeness analysis are novel applications of the domain, presented for the first time in this chapter.

To conclude the chapter, 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 $\beta_1 = \{\langle \oplus = \oplus x, \text{One} \rangle, \langle 0 < \oplus x, \text{One} \rangle\}$. It is easy to see that θ_1 is satisfiable (i.e. $\text{Cis_sat}(\theta_1)$ is true). Also we have that $\text{Ais_sat}(\beta_1)$ 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 $\beta_2 = \{\langle \oplus = \oplus x, \text{One} \rangle, \langle \oplus < \oplus x, \text{One} \rangle\}$. It is easy to see that $\text{Cis_sat}(\theta_2)$ is true, however $\text{Ais_sat}(\beta_2)$ 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 unable to remain accurate in the second example because grouping all the positive numbers into one equivalence set is too coarse an abstraction. 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 chapter.

Chapter 4

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 chapter is organized as follows. Section 4.1 introduces the abstract objects of the domain `LInt` and their concretization. Section 4.2 briefly presents the operations and applications of the domain. The power domain and the widening are presented in Section 4.3. The chapter then presents a complete worked example in Section 4.4. The chapter concludes in Section 4.5 by discussing how `LInt` addresses the limitations of `LSign`. The proofs of the results in this chapter can be found in Appendix A.

4.1 Abstract Objects and Concretization

We consider the set of real numbers \mathbb{R} and its extension $\mathbb{R}^\infty = \mathbb{R} \cup \{-\infty, \infty\}$. The ordering on \mathbb{R} is extended to \mathbb{R}^∞ , as usual, by

$$\forall r \in \mathbb{R} : -\infty < r < +\infty.$$

We also consider a finite subset \mathcal{F} of \mathbb{R} 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}^∞ . We now define the intervals more formally, as well as what they represent.

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

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

The monotone concretization function $Cc : \text{Intv} \rightarrow 2^{\mathbb{R}}$ is defined as

$$\begin{cases} Cc([a_\ell, a_r]) &= \{c \mid c \in \mathbb{R} \wedge a_\ell \leq c \leq a_r\} \\ Cc((a_\ell, a_r]) &= \{c \mid c \in \mathbb{R} \wedge a_\ell < c \leq a_r\} \\ Cc([a_\ell, a_r)) &= \{c \mid c \in \mathbb{R} \wedge a_\ell \leq c < a_r\} \\ Cc((a_\ell, a_r)) &= \{c \mid c \in \mathbb{R} \wedge a_\ell < c < a_r\} \end{cases}$$

Intervals are denoted by the letter s , possible subscripted.

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

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

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

The ordering $\sqsubseteq : \text{Intv} \times \text{Intv}$ is defined by

$$s_1 \sqsubseteq s_2 \Leftrightarrow \neg \text{increase}(s_1, s_2) \wedge \neg \text{decrease}(s_1, s_2)$$

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, that are consistent approximations of the corresponding operations on \mathbb{R}^∞ . More formally,

$$c_1 \in Cc(s_1) \wedge c_2 \in Cc(s_2) \Rightarrow c_1 + c_2 \in Cc(s_1 + s_2)$$

$$c_1 \in Cc(s_1) \wedge c_2 \in Cc(s_2) \Rightarrow c_1 - c_2 \in Cc(s_1 - s_2)$$

$$c_1 \in Cc(s_1) \wedge c_2 \in Cc(s_2) \Rightarrow c_1 \times c_2 \in Cc(s_1 \times s_2)$$

$$c \in Cc(s) \Rightarrow -c \in Cc(-s)$$

$$c \in Cc(s_1) \wedge c \in Cc(s_2) \Rightarrow c \in Cc(s_1 \sqcap s_2)$$

$$c \in Cc(s_1) \vee c \in Cc(s_2) \Rightarrow c \in Cc(s_1 \sqcup s_2)$$

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

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

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

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

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 32 The abstract constraint $(-\infty, +\infty) = [1, 1]P + [1, 1]R$ represents both the constraint $3 = P + R$ and $-3 = P + R$ but not the constraint $3 = 2P + 3R$.

Example 33 The abstract constraint with multiplicity $\langle (-\infty, +\infty) = [1, 1]P + [1, 1]R, \text{One} \rangle$ represents only multisets of size 1, e.g., $\{3 = P + R\}$. $\langle 0 \leq [1.01, +\infty)P + (-\infty, -1]R, \text{Any} \rangle$ represents multisets of any size, e.g., \emptyset , $\{0 \leq 3P - R\}$ and $\{0 \leq 3P - R, 0 \leq 2P - 3R\}$.

Example 34 The abstract store $\beta = \{ \langle (-\infty, +\infty) = [1, 1]P + [1, 1]R, \text{One} \rangle, \langle 0 \leq [1.01, +\infty)P + (-\infty, -1]R, \text{Any} \rangle \}$ represents constraint stores with at least one constraint, and their equivalence classes. For example $\{3 = P + R\} \in Cc_i(\beta)$ and $\{3 = P + R, 0 \leq 2P - 3R\} \in Cc_i(\beta)$. Further, $\{3 = P + R, 9 \leq 5P\} \in Cc(\beta)$ because $\{3 = P + R, 9 \leq 5P\} \leftrightarrow \{3 = P + R, 0 \leq 2P - 3R\}$.

```

Asplit_top( $\beta, v$ ) {
  if  $\beta = \emptyset$  return  $\emptyset$ ;
  else let  $\beta = \{\gamma\} \cup \beta' \wedge \gamma = \langle \sigma, \mu \rangle \notin \beta'$ 
     $\gamma^0 = \langle \sigma^0, \mu \sqcup \text{ZeroOrOne} \rangle$ 
     $\gamma^+ = \langle \sigma^+, \mu \sqcup \text{ZeroOrOne} \rangle$ 
     $\gamma^- = \langle \sigma^-, \mu \sqcup \text{ZeroOrOne} \rangle$ 
     $\sigma^0 = \sigma$  except that  $\sigma^0[v] = [0, 0]$ 
     $\sigma^+ = \sigma$  except that  $\sigma^+[v] = \sigma[v] \cap (0, +\infty)$ 
     $\sigma^- = \sigma$  except that  $\sigma^-[v] = \sigma[v] \cap (-\infty, 0)$  in
    case not ( $\text{pos}(\sigma[v]) \vee \text{neg}(\sigma[v]) \vee \text{zer}(\sigma[v])$ ):
       $\beta'' := \emptyset$ 
      if contains_zer( $\sigma[v]$ )
         $\beta'' := \beta'' \uplus \gamma^0$ ;
      if contains_pos( $\sigma[v]$ )
         $\beta'' := \beta'' \uplus \gamma^+$ ;
      if contains_neg( $\sigma[v]$ )
         $\beta'' := \beta'' \uplus \gamma^-$ ;
      return Asplit_top( $\beta', v$ )  $\uplus$   $\beta''$ ;
    otherwise:
      return Asplit_top( $\beta', v$ )  $\uplus$   $\gamma$ ;
}

```

Figure 4.1: Modified Asplit_top Algorithm for LInt

4.2 Operations and Applications

The algorithms for various operations (ordering, addition of a constraint, upper bound and projection) as well as the applications (satisfiability, conditional satisfiability, redundancy and freeness) 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 $\text{pos}(\sigma[v])$, while an assignment $\sigma[v] := 0$ would be replaced by the assignment $\sigma[v] := [0, 0]$. The only algorithm that undergoes a non-trivial change is the algorithm for operation Asplit_top. It can be more precise due to the more precise information available about the coefficient of the variable being eliminated. The algorithm for Asplit_top in LInt is given in Figure 4.1.

4.3 The Power Domain 2^{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^{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^{LInt} completely parallel the corresponding definitions in the domain 2^{LSign} . However, as 2^{LInt} is an infinite domain, it becomes necessary to introduce a *widening* operator¹. The use of widening operators was proposed in [10] and has been previously used in domains such as type graphs for Prolog [50, 51].

The design of a widening operator is both experimental and theoretical in nature. On 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. We begin by presenting the ideas behind the widening operator informally, and then present the operator in a more formal setting.

4.3.1 Informal Presentation

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

1. when updating the output description of a predicate;
2. when updating the input description of a predicate;
3. when updating the intermediate 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^{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.

¹In fact, it is necessary to define a widening for the LInt domain as well, but we shall focus on the 2^{LInt} domain, as it is used in practice.

The situation is not so simple in the domain 2^{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 intuition is to curtail the growth in the number of abstract stores in the description by placing some restrictions. The restrictions should be such that the domain does not lose information that is “interesting” to the application. The approach taken is to require that

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^{LInt} multistore produces a valid LSign multistore (i.e. no duplicate stores).

This enables a natural mapping of LInt abstract objects to LSign abstract objects and automatically restricts the cardinality of an 2^{LInt} multistore to a finite number, because there are only a finite number of LSign stores over a given domain of variables. Placing these restrictions on 2^{LInt} abstract objects guarantees that a 2^{LInt} abstract description cannot keep increasing in size (i.e. having more and more LInt stores in it). This is the first component of the widening.

However there is one more way in which a sequence of 2^{LInt} abstract multistores can grow indefinitely. Even if the number of LInt stores in the description becomes fixed, the intervals of those stores can be refined infinitely often. The intuition behind the solution to this problem is to accelerate the refinements of the intervals such that the acceleration reflects the way in which the intervals are changing and the acceleration causes the number of refinements to be finite. The approach taken is to guarantee that if the number of LInt stores in the description is not increasing, then the number of zero or infinity symbols in the description increases. As there can only be a finite number of zero or infinity symbols in the description, the sequence of refinements must terminate. This is the second component of the widening. In order to simplify the widening, it is necessary to require that the second component (i.e. accelerating the change of the intervals of the LInt stores), be applied only to stores that are the same (modulo replacement of intervals by signs).

We now explain the above with the help of an example. 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. In other words,

we primarily concern ourselves with the second component of the widening and leave the first component for the formal presentation. 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 .

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

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

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

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 $\langle [3, 5] = [1, 1]T, One \rangle$ and $\langle [3, 6] = [1, 1]T, One \rangle$. 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 $\langle [3, +\infty) = [1, 1]T, One \rangle$. 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.

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

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.

4.3.2 Formal Presentation

We now formally present the widening operator for the domain 2^{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.

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

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

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

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

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

- (i) $\text{shape}(s_{old}) = \text{shape}(s_{new} \nabla s_{old})$.
- (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

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

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. $\text{shape}(\gamma_{new}) = \text{shape}(\gamma_{old})$. 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 16 Let γ_{old} and γ_{new} be abstract constraints with multiplicity s.t. $\text{shape}(\gamma_{new}) = \text{shape}(\gamma_{old})$. Then

²It is possible to be more accurate and instead use the largest positive constant in the program that is smaller than the left endpoint.

- (i) $\text{shape}(\gamma_{old}) = \text{shape}(\gamma_{new} \nabla \gamma_{old})$.
- (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

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

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

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

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

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

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

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

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

- (i) $\text{shape}(\beta_{old}) = \text{shape}(\beta_{new} \nabla \beta_{old})$ 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
- $\forall \beta_1, \beta_2 \in \alpha : \text{shape}(\beta_1) \neq \text{shape}(\beta_2)$

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

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

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

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

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

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

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

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

The following theorem states the correctness of the operator ∇ .

Theorem 15 [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 18 (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 $\alpha'_0, \dots, \alpha'_i, \dots$ be a sequence of abstract multistores and $\alpha_0, \dots, \alpha_i, \dots$ a sequence of normalized abstract multistores defined as

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

The proof consists of showing that $\alpha_0, \dots, \alpha_i, \dots$ 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. \square

4.4 Complete Example

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

Example 35 [Computation of LInt Output Description for `mg/4`] The abstract substitution describing the output of the first clause is

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

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

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

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

$$\{\{\langle [0, 0] < [1, 1]T, \text{One} \rangle, \langle [0, 0] \leq [1, 1]P, \text{One} \rangle, \\ \langle [0, 0] = [1.01, 1.01]P + [-1, -1]R + [-1, -1]P1, \text{One} \rangle, \langle [1, 1] = [1, 1]T + [-1, -1]T1, \text{One} \rangle, \\ \langle [0, 0] = [1, 1]T1, \text{One} \rangle, \langle [0, 0] = [1, 1]P1 + [-1, -1]B, \text{One} \rangle\}\}$$

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

$$\{\{\langle [0, 0] < [1, 1]T, \text{One} \rangle, \langle [1, 1] = [1, 1]T, \text{One} \rangle, \\ \langle [0, 0] \leq [1, 1]P, \text{One} \rangle, \langle [0, 0] = [1.01, 1.01]P + [-1, -1]R + [-1, -1]B, \text{One} \rangle\}\}$$

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*.

$$\{ \\ \{\langle [0, 0] = [1, 1]T, \text{One} \rangle, \langle [0, 0] = [1, 1]P + [-1, -1]B, \text{One} \rangle\}, \\ \{\langle [0, 0] < [1, 1]T, \text{One} \rangle, \langle [1, 1] = [1, 1]T, \text{One} \rangle, \\ \langle [0, 0] \leq [1, 1]P, \text{One} \rangle, \langle [0, 0] = [1.01, 1.01]P + [-1, -1]R + [-1, -1]B, \text{One} \rangle\} \\ \}$$

In this case, the widening is just the union of multistores 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}):

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

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

```
{
  {⟨[0, 0] = [1, 1]T, One⟩, ⟨[0, 0] = [1, 1]P + [-1, -1]B, One⟩},
  {⟨[0, 0] < [1, 1]T, One⟩, ⟨[1, 1] = [1, 1]T, One⟩,
    ⟨[0, 0] ≤ [1, 1]P, One⟩, ⟨[0, 0] = [1.01, 1.01]P + [-1, -1]R + [-1, -1]B, One⟩}
  {⟨[0, 0] < [1, 1]T, One⟩, ⟨[0, 0] ≤ [1, 1]P, One⟩, ⟨[2, 3] = [1, 1]T, One⟩,
    ⟨[0, 0] = [1.0201, 103.0301]P + [-303.01, -2.01]R + [-100, -1]B, One⟩,
    ⟨[0, 0] ≤ [1, 201]R + [1, 100]B, One⟩, ⟨[0, 0] ≤ [1, 201]R + [1, 100]B, Any⟩},
  {⟨[0, 0] < [1, 1]T, One⟩, ⟨[0, 0] ≤ [1, 1]P, One⟩, ⟨[3, 6] = [1, 1]T, One⟩,
    ⟨[0, 0] = [1.030301, 1061520.146432]P + [-6152015.314944, -3.0301]R
      + [-999999.995904, -1]B, One⟩,
    ⟨[0, 0] ≤ [1, 201]R + [1, 100]B, One⟩, ⟨[0, 0] ≤ [1, 201]R + [1, 100]B, Any⟩,
    ⟨[0, 0] ≤ [1.01, 10303.01]P + [-30301, -1]R, One⟩,
    ⟨[0, 0] ≤ [1.01, 10303.01]P + [-30301, -1]R, Any⟩}
}
```

At this point the widening needs to be applied and produces the following multistore as the updated output of mg ($\alpha_{new} \nabla \alpha_{old}$):

```
{
  {⟨[0, 0] = [1, 1]T, One⟩, ⟨[0, 0] = [1, 1]P + [-1, -1]B, One⟩},
  {⟨[0, 0] < [1, 1]T, One⟩, ⟨[1, 1] = [1, 1]T, One⟩,
    ⟨[0, 0] ≤ [1, 1]P, One⟩, ⟨[0, 0] = [1.01, 1.01]P + [-1, -1]R + [-1, -1]B, One⟩}
  {⟨[0, 0] < [1, 1]T, One⟩, ⟨[0, 0] ≤ [1, 1]P, One⟩, ⟨[2, 3] = [1, 1]T, One⟩,
    ⟨[0, 0] = [1.0201, 103.0301]P + [-303.01, -2.01]R + [-100, -1]B, One⟩,
    ⟨[0, 0] ≤ [1, 201]R + [1, 100]B, One⟩, ⟨[0, 0] ≤ [1, 201]R + [1, 100]B, Any⟩},
  {⟨[0, 0] < [1, 1]T, One⟩, ⟨[0, 0] ≤ [1, 1]P, One⟩, ⟨[3, +∞) = [1, 1]T, One⟩,
    ⟨[0, 0] = [1.030301, +∞)P + (-∞, -3.0301]R + (-∞, -1]B, One⟩,
    ⟨[0, 0] ≤ [1, 201]R + [1, 100]B, One⟩, ⟨[0, 0] ≤ [1, 201]R + [1, 100]B, Any⟩,
    ⟨[0, 0] ≤ [1.01, +∞)P + (-∞, -1]R, One⟩,
    ⟨[0, 0] ≤ [1.01, +∞)P + (-∞, -1]R, Any⟩}
}
```

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 36 [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

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

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

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

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

$$\begin{aligned} &\{ \langle [-1, -1] < [1, 1]T1, \text{One} \rangle, \langle (-\infty, +\infty) = [1, 1]R, \text{One} \rangle, \\ &\quad \langle (-\infty, +\infty) = [1, 1]P1 + [1, 1]R, \text{One} \rangle \} \end{aligned}$$

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

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

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

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

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 $mg(\alpha_{old})$:

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

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

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

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

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

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.

4.5 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 chapter. 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 redundancy and conditional satisfiability.

Chapter 5

Abstract Domain Prop

In this chapter, we present the abstract domain `Prop` [35, 52] which is useful to deduce groundness information in Prolog programs. It can also be extended easily to infer fixed variables in CLP(eg. [17]). The presentation is basically a review of material already presented in [52], along with the extension of `Prop` to handle the constraints of $\text{CLP}(\mathcal{R}_{Lin})$. Section 5.1 defines the abstract objects and their concretization. The abstract operations of the domain are presented in Section 5.2. The application of the domain to deduce fixed variables is discussed in Section 5.3. The chapter concludes with a complete worked example in Section 5.4.

5.1 Abstract Objects and Concretization

The key idea of `Prop` is the abstraction of the objects in the concrete domain by appropriate Boolean formulas. The domain and its ordering can be defined more formally as follows.

Definition 29 [Abstract Domain] The domain `Prop` over $D = \{x_1, \dots, x_n\}$, denoted Prop_D , is the poset of Boolean functions represented by propositional formulas constructed from D , the Boolean truth values (true and false) and the logical connectives. The domain is ordered by implication. Boolean functions are denoted by the letter f .

In order to define the concretization, it is necessary to define the following concepts.

Definition 30 [Truth Assignment] A truth assignment over D is as a function $I : D \rightarrow \text{Boolean}$. The value of a Boolean function f w.r.t. a truth assignment I is denoted $I(f)$.

When $I(f)$ is true, we say that I satisfies f .

The intuition behind the domain **Prop** is that a concrete constraint store θ is abstracted by a Boolean function f over D if and only if the truth assignment I defined by

$$I(x_i) = \text{true} \Leftrightarrow x_i \text{ is fixed in } \theta \quad (1 \leq i \leq n)$$

satisfies f . For example $x_1 \leftrightarrow x_2$ abstracts the stores $\{x_1 = x_2 + 1\}$ and $\{x_1 = 1, x_2 = 4\}$ but not the store $\{x_1 = 3, x_2 < x_1\}$.

Definition 31 [Concretization Function] The concretization function $Cc : \text{Prop}_D \rightarrow CS_D$ is defined as

$$Cc(f) = \{\theta \mid (assign(\theta))(f) = \text{true}\}$$

where $assign : CS_D \rightarrow D \rightarrow \text{Boolean}$ is defined by $assign \theta x_i = \text{true} \Leftrightarrow x_i$ is fixed in θ .

Prop_D is a finite lattice, where the greatest lower bound is given by the conjunction and the least upper bound is given by the disjunction.

Definition 32 [Valuation] The valuation of a Boolean function f w.r.t. a variable x_i and a Boolean truth value b is the function obtained by replacing x_i by b in f , and is denoted as $f|_{x_i=b}$.

5.2 Abstract Operations

We now show how the operations of the abstract framework such as **RESTRC**, **AI_VAR**, **EXTG** etc. can be defined for the domain **Prop**.

UNION (Upper bound) Operation **UNION** is used to compute the result of a predicate, given the result of its individual clauses. Its abstract version is just the disjunction for Boolean formulas.

$$\text{UNION}(f_1, \dots, f_n) = f_1 \vee \dots \vee f_n$$

AI_ADD (Addition of a constraint) Operation **AI_ADD** is used when a constraint is encountered in the program. Its abstract version adds the abstraction of the constraint to the input abstract substitution. Let the concrete constraint λ in the program be

$c_0 \delta \sum_{i=1}^n c_i x_i$ (where $c_i \neq 0$ for $1 \leq i \leq n$). Then

$$\text{AI_ADD}(\lambda, f) = \begin{cases} f & \text{if } \delta \in \{<, \leq, \neq\} \\ f \wedge x_1 & \text{if } \delta \text{ is '=' and } n = 1 \\ f \wedge (x_1 \wedge \dots \wedge x_{n-1} \rightarrow x_n) \\ \quad \wedge (x_2 \wedge \dots \wedge x_n \rightarrow x_{n-1}) \\ \quad \vdots \\ \quad \wedge (x_2 \wedge \dots \wedge x_n \rightarrow x_1) & \text{if } \delta \text{ is '=' and } n > 1 \end{cases}$$

RESTRC (Restriction of a clause substitution) Operation RESTRC is used at the end of a clause execution to restrict the substitution expressed on all the clause variables to a substitution expressed on only the head variables of the clause. Its abstract version eliminates the variables that occur in the clause body but not in the clause head. Let x_{m+1}, \dots, x_n be the variables that occur only in the body of c . Then

$$\text{RESTRC}(c, f) = \text{elim_all } \{x_{m+1}, \dots, x_n\} f$$

where

$$\begin{aligned} \text{elim_all } \emptyset f &= f \\ \text{elim_all } \{x_j, \dots, x_n\} f &= \\ \text{elim_all } \{x_{j+1}, \dots, x_n\} (f|_{x_j=\text{true}} \vee f|_{x_j=\text{false}}) &\quad (m < j \leq n) \end{aligned}$$

EXTC (Extension of a clause substitution) Operation EXTC is used to is used at the beginning of a clause's execution to express the input substitution on all the clause variables and not just the head variables. Its abstract version for a clause c is a trivial operation.

$$\text{EXTC}(c, f) = f$$

RESTRG (Restriction of a substitution before a literal) Operation RESTRG is used before the execution of a literal in the body of a clause to compute the input substitution for the execution of the literal. Its abstract version first projects out the variables of the clause that do not appear in the literal from the abstract substitution, and then normalizes the resulting abstract substitution. Let x_{i_1}, \dots, x_{i_m} be the variables that occur in the literal l and let x_1, \dots, x_n be the variables of the clause in which the literal occurs. Then

$$\text{RESTRG}(l, f) = \text{norm } [x_{i_1}, \dots, x_{i_m}] (\text{elim_all } (\{x_1, \dots, x_n\} \setminus \{x_{i_1}, \dots, x_{i_m}\}) f)$$

EXTG (Extension of a substitution after a literal) Operation EXTG is used after the execution of a literal to extend its result to the clause substitution. Its abstract version first denormalizes the abstract substitution f' for the output of the literal l and then adds it to the clause abstract substitution f . Let x_{i_1}, \dots, x_{i_m} be the variables that appear in the literal l , in that order. Then

$$\text{EXTG}(l, f, f') = f \wedge \text{denorm}[x_{i_1}, \dots, x_{i_m}] f'$$

5.3 Application

The only application of the domain Prop that is of interest to us is its use to determine fixed or ground variables. The function $\text{Ais_fixed} : \text{Prop}_D \times D \rightarrow \text{Boolean}$ takes a Prop description f over the domain D and a variable $x \in D$ and returns true if the variable can be deduced to be fixed in every constraint store in the concretization of β . The implementation of Ais_fixed is straightforward:

$$\text{Ais_fixed}(f, x) = (f \rightarrow x)$$

and it satisfies the following specification:

Specification 20 $\text{Ais_fixed} : \text{Prop}_D \times D \rightarrow \text{Boolean}$ should satisfy the following consistency condition.

$$\forall \theta \in CS_D, \forall f \in \text{Prop}_D, \forall x \in D : \theta \in Cc(f) \Rightarrow (\text{Ais_fixed}(f, x) \Rightarrow x \text{ is fixed in } \theta)$$

Example 37 Consider a call to the predicate `mg/4` whose input can be described in the Prop domain by the following abstract substitution

$$P \wedge R$$

We have that

$$\text{Ais_fixed}(P \wedge R, P) = (P \wedge R \rightarrow P) = \text{true}$$

indicating that P is always fixed before the execution of $P \geq 0$.

5.4 Complete Example

We now present a complete example that shows the computation of the output and input descriptions in the domain Prop, for the mortgage program `mg/4`. First we present the computation of the goal independent (or online) output.

Example 38 [Computation of Prop Output Description for $mg/4$] The abstract substitution describing the output of the first clause is

$$T \wedge (B \leftrightarrow P)$$

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

$$(P1 \wedge P \rightarrow R) \wedge (P \wedge R \rightarrow P1) \wedge (P1 \wedge R \rightarrow P) \wedge (T \leftrightarrow T1)$$

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

$$(P1 \wedge P \rightarrow R) \wedge (P \wedge R \rightarrow P1) \wedge (P1 \wedge R \rightarrow P) \wedge (T \leftrightarrow T1) \wedge T1 \wedge (B \leftrightarrow P1)$$

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

$$T \wedge (B \wedge P \rightarrow R) \wedge (P \wedge R \rightarrow B) \wedge (B \wedge R \rightarrow P)$$

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

$$T \wedge (P \wedge R \rightarrow B) \wedge (B \wedge R \rightarrow P)$$

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

$$(P1 \wedge P \rightarrow R) \wedge (P \wedge R \rightarrow P1) \wedge (P1 \wedge R \rightarrow P) \wedge (T \leftrightarrow T1) \wedge T1 \wedge (P1 \wedge R \rightarrow B) \wedge (B \wedge R \rightarrow P1)$$

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

$$T \wedge (P \wedge R \rightarrow B) \wedge (B \wedge R \rightarrow P)$$

which is the fixpoint for the computation of the output of mg .

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

Example 39 [Computation of Prop Input Description for $mg/4$] The abstract substitution describing a top level query such that P and R are fixed, is given as

$$P \wedge R$$

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

$$P \wedge R \wedge (P1 \wedge P \rightarrow R) \wedge (P \wedge R \rightarrow P1) \wedge (P1 \wedge R \rightarrow P) \wedge (T \leftrightarrow T1)$$

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

$$P1 \wedge R$$

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

$$P \wedge R$$

This is the fixpoint of the input computation for `mg`, indicating that `P` and `R` are fixed not only for the top level query, but also for all recursive calls to `mg`.

Chapter 6

Program Transformations

In this chapter, we present the various program transformations (or optimizations) proposed for $\text{CLP}(\mathcal{R}_{Lin})$ programs. The chapter is organized as follows. Section 6.1 gives a formal presentation for the reordering optimization including a proof of correctness. Sections 6.2 and 6.3 discuss removal and refinement informally. Section 6.4 describes how the optimizations are integrated in the optimizing compiler. The detailed proofs of the results in this chapter may be found in Appendix A.

6.1 Reordering Optimization

As mentioned previously in Section 1.5, the purpose of reordering is to move constraints towards the end of the clauses so that they can be specialized into assignments and tests. However, reordering can take place only if it preserves the same search space which provides guarantees that the reordered program will be at least as efficient (asymptotically) as the original program. In general in the literature (see for example [37]), this issue is only discussed informally. It is typically mentioned that two goals g_1 and g_2 can be reordered in the context of a constraint store θ if, for any satisfiable store θ_1 occurring during the execution of g_1 with input store θ and any satisfiable store θ_2 occurring during the execution of g_2 with input store θ , $\theta_1 \wedge \theta_2$ is satisfiable. However, this leaves open a number of delicate issues such as whether to use the original or the reordered program when computing the intermediate stores of g_1 and g_2 . It turns out that answers to these questions are in fact more complicated than expected and really deserve to be addressed properly.

The purpose of this section is to describe the reordering algorithm in detail and to prove its correctness. It is organized as follows. Section 6.1.1 gives a modified syntax for $\text{CLP}(\mathcal{R}_{Lin})$ programs that we shall use in this chapter to simplify the proofs. Section 6.1.2 presents an operational semantics of the language which is similar to the standard semantics based on SLD-Resolution with a left-to-right selection rule [30] but adapted to simplify our proofs. Section 6.1.3 formulates the reordering problem and defines formally which reorderings are admissible, i.e., which reorderings preserve the search space. The definition of admissible reorderings is not easy to approximate; Section 6.1.4 defines the notion of failure-free reorderings and shows that failure-free reorderings are admissible, the main result of this section. Failure-free reorderings are based on a condition which is much more amenable to abstraction and is the basis of our transformations. Section 6.1.5 describes how failure-free reorderings may be abstracted. Section 6.1.6 gives a complete example of the reordering optimization.

6.1.1 Modified Syntax

The syntax of $\text{CLP}(\mathcal{R}_{Lin})$ programs that we use in this section is a slightly modified and simplified version of the standard syntax given in Chapter 1. As mentioned, the modified syntax simplifies the presentation and proofs in the rest of the section. The modified syntax for $\text{CLP}(\mathcal{R}_{Lin})$ programs is essentially the same as the original syntax, the only differences being

1. a sequence of adjacent constraints in the body of a clause that are not separated by a predicate are grouped together into a (possibly empty) multiset (akin to a *constraint store*);
2. the constraint store before a predicate in the body of the clause is paired together with the predicate in a tuple.

Figure 6.1 shows an outline of the modified syntax of a $\text{CLP}(\mathcal{R}_{Lin})$ program. A $\text{CLP}(\mathcal{R}_{Lin})$ 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 $p(t_1, \dots, t_n)$ where t_1, \dots, t_n are terms. A term is a variable (e.g. X). Note that the language is restricted to not allow functors. A body is either ϵ (the empty body) or a sequence of goals. Each goal is a tuple that contains a store as the first element and an atom as the second element. In order to accomodate trailing constraints in the body, we assume that every program contains a trivially satisfiable clause ($\text{true} : - \epsilon$) whose head can be used as a dummy

```

Program ::= Clauses
Clauses ::=  $\epsilon$  | Clause Clauses
Clause  ::= Head :- Body.
Head    ::= Atom
Body    ::=  $\epsilon$  | Goal :: Body
Goal    ::=  $\langle \text{Store}, \text{Atom} \rangle$ 
Store   ::=  $\emptyset$  | Store  $\sqcup$  {Constraint}

```

Figure 6.1: Outline of the Modified Syntax of $\text{CLP}(\mathcal{R}_{Lin})$.

atom in the goal for the trailing constraints. Sometimes, the body $b_1 :: \dots :: b_n$ may be rewritten as b_1, \dots, b_n . in order to simplify the notation. Also clauses of the form $H : - \epsilon$ (i.e with an empty body) may be rewritten as $H..$ These are merely syntactic sugar. A store is a (possibly empty) multiset of constraints. The constraints of $\text{CLP}(\mathcal{R}_{Lin})$ are real linear constraints.

Example 40 The clause

$$top : - \lambda_1, \lambda_2, p, q, \lambda_3, r, \lambda_4.$$

when rewritten in the modified syntax appears as

$$top : - \langle \{\lambda_1, \lambda_2\}, p \rangle, \langle \emptyset, q \rangle, \langle \{\lambda_3\}, r \rangle, \langle \{\lambda_4\}, \text{true} \rangle.$$

6.1.2 Concepts from Operational Semantics

This section defines the necessary concepts from operational semantics to justify the optimizations. As mentioned before, most of these concepts are variations of standard concepts in the theory of constraint logic programming. A computation state represents a snapshot of the execution and contains the constraints accumulated so far and the goals which remain to execute.

Definition 33 [Computation State] A computation state is syntactically a tuple whose first element is a Body and whose second element is a Store.

$$\text{State} ::= \langle \text{Body} \diamond \text{Store} \rangle.$$

Computation states are denoted by the letter S possibly subscripted. The shape of a computation state $\langle G \diamond \theta \rangle$ is its goal part, i.e.,

$$\text{shape}(\langle G \diamond \theta \rangle) = G.$$

The store of the computation state is simply its constraint store

$$\text{stores}(\langle G \Diamond \theta \rangle) = \{\theta\}.$$

As usual, computation steps are moves from a computation state to another. We now define a notion of similarity between computation states which ignores constraint stores. This is the first of a series of definitions formalizing what it means to preserve the search space.

Definition 34 [Similar Computation States] Two states are said to be similar (denoted by \approx) if they have the same shape, i.e.

$$S_1 \approx S_2 \text{ if } \text{shape}(S_1) \equiv \text{shape}(S_2).$$

The notion of a computation tree is used to represent the search space. A computation tree should be visualized as a tree labelled with a computation state and whose children are all the computation trees of the computation states obtained by applying a computation step. The definition below is purely syntactical. The operational semantics defined subsequently shows how to build a computation tree for a given program and query.

Definition 35 [Computation Tree] A computation tree is a record whose first element (the root of the tree) is a computation state and whose second element (the children of the root) is a list of trees.

$$\text{Tree} ::= \text{tree}(\text{State}, \text{ListTree})$$

$$\text{ListTree} ::= [] \mid [\text{Tree} \mid \text{ListTree}]$$

Computation trees are denoted by the letter T possibly subscripted, while lists of trees are denoted by the letter L . If T is the tree $\text{tree}(S, L)$, then S is called the *label* of the tree, while L is called the *child list* of the tree.

The shape of a computation tree is fundamental in comparing the search space of two programs.

Definition 36 [Shape of a Computation Tree] The shape of a tree is defined as follows:

$$\text{shape}(\text{tree}(S, L)) = \text{tree}(\text{shape}(S), \text{shape}(L)).$$

$$\text{shape}([]) = [].$$

$$\text{shape}([T \mid L]) = [\text{shape}(T) \mid \text{shape}(L)].$$

The stores of a computation are fundamental in defining our main criterion to decide whether a reordering preserves the same search tree.

Definition 37 [Stores of a Computation Tree] The stores of a tree are defined as follows.

$$\text{stores}(\text{tree}(S, L)) = \text{stores}(S) \cup \text{stores}(L).$$

$$\text{stores}([\]) = \emptyset.$$

$$\text{stores}([T \mid L]) = \text{stores}(T) \cup \text{stores}(L).$$

We now define what it means for two trees to be similar up to a certain depth.

Definition 38 [Similar Trees] Let T_1 and T_2 be the trees $\text{tree}(S_1, L_1)$ and $\text{tree}(S_2, L_2)$ respectively. T_1 and T_2 are similar up to depth d , denoted by $T_1 \approx_d T_2$, when the following conditions are satisfied.

$$T_1 \approx_d T_2 = \begin{cases} S_1 \approx S_2 & \text{if } d = 1 \\ S_1 \approx S_2 \wedge L_1 \approx_{d-1} L_2 & \text{otherwise} \end{cases}$$

where

$$L_1 \approx_d L_2 = \begin{cases} \text{true} & \text{if } L_1 = L_2 = [\] \\ T_1 \approx_d T_2 \wedge L_3 \approx_d L_4 & \text{if } L_1 = [T_1 \mid L_3] \text{ and } L_2 = [T_2 \mid L_4] \\ \text{false} & \text{otherwise} \end{cases}$$

Moreover, two trees are similar (denoted $T_1 \approx T_2$) if, for any finite depth d ,

$$T_1 \approx_d T_2.$$

We are now ready to define the operational semantics used in this chapter to prove the correctness of reordering. The semantics receives as input a computation state and a program and constructs a computation tree representing all the computations that starts at that state using a left-to-right selection rule.

Definition 39 [Operational Semantics $\tau_P : \text{State} \mapsto \text{Tree}$] The operational semantics of a program P is a mapping $\tau_P : \text{State} \mapsto \text{Tree}$ defined as follows:

$$\tau_P(\langle G \diamond \theta \rangle) = \text{tree}(\langle G \diamond \theta \rangle, L)$$

where

$$L = \begin{cases} [] & \text{if } G = \epsilon \\ [] & \text{if } G = \langle \theta', q \rangle :: B \text{ and} \\ & \theta \sqcup \theta' \text{ is inconsistent or no clause in } P \text{ has head } q \\ [T_1, \dots, T_n] & \text{if } G = \langle \theta', q \rangle :: B \text{ and } \theta \sqcup \theta' \text{ is consistent and} \\ & T_i = \tau_P(\langle B_i :: B \Diamond \theta \sqcup \theta' \sqcup (q = H_i) \rangle) \text{ where } H_i : -B_i \text{ is the } i^{th} \text{ of} \\ & \text{the } n \text{ clauses of } P \text{ with head } q \text{ (renamed with all new variables)} \\ & \text{and } q = H_i \text{ adds the equality constraints between the actual} \\ & \text{parameters of } q \text{ and formal parameters of } H_i \end{cases}$$

Note that if $\theta \sqcup \theta'$ is known to be consistent, then $\theta \sqcup \theta' \sqcup (q = H_i)$ is also consistent because $q = H_i$ only introduces simple equality constraints between the actual parameters of q and the formal parameters of H_i which are all new variables.

Example 41 Consider the mortgage example

$\text{mg}(P, T, R, B) :- \langle \{T = 0, B = P\}, \text{true} \rangle.$

$\text{mg}(P, T, R, B) :- \langle \{T > 0, P \geq 0, P1 = P * 1.01 - R, T1 = T - 1\}, \text{mg}(P1, T1, R, B) \rangle.$

Given a query $S = \langle \langle \emptyset, \text{mg}(P, T, R, B) \rangle \Diamond \{P = 1, R = 0.01\} \rangle$, the computation tree for this query is given by $\tau_{\text{mg}}(S) = \text{tree}(S, [T_1, T_2])$, where

$$\begin{aligned} T_1 &= \tau_{\text{mg}}(\langle \langle \{T' = 0, B' = P'\}, \text{true} \rangle \Diamond \{P = 1, R = 0.01, P = P', T = T', R = R', B = B'\} \rangle) \\ T_2 &= \tau_{\text{mg}}(\langle \langle \{T' > 0, P' \geq 0, P1 = P' * 1.01 - R', T1 = T' - 1\}, \text{mg}(P1, T1, R', B') \rangle \Diamond \\ &\quad \{P = 1, R = 0.01, P = P', T = T', R = R', B = B'\} \rangle) \end{aligned}$$

6.1.3 Admissible Reorderings

As mentioned previously, only reorderings preserving the search space should be performed by an optimizing compiler. The purpose of this section is to formalize this notion by defining *admissible* reorderings. We start by defining the reorderings considered in our optimizer.

Definition 40 [Simple Reordering] Let P be a program consisting of the sequence of clauses $c_1, \dots, c_i, \dots, c_n$, where the clause c_i is

$$p : -\langle \theta_1, q_1 \rangle, \dots, \langle \theta_k \sqcup \{\lambda\}, q_k \rangle, \langle \theta_{k+1}, q_{k+1} \rangle, \dots, \langle \theta_m, q_m \rangle.$$

Let P' be the program consisting of the sequence of clauses $c_1, \dots, c'_i, \dots, c_n$, where the clause c'_i is

$$p : -\langle \theta_1, q_1 \rangle, \dots, \langle \theta_k, q_k \rangle, \langle \theta_{k+1} \sqcup \{\lambda\}, q_{k+1} \rangle, \dots, \langle \theta_m, q_m \rangle.$$

Then P' is a simple reordering of P .

A simple reordering is obtained by moving one constraint over exactly one predicate call. The reordering relation between programs is just the transitive closure of the simple reordering relation.

Definition 41 [Reordering] P' is a reordering of P if

1. P' is a simple reordering of P ; or
2. there is a program P'' such that P'' is a reordering of P and P' is a reordering of P'' .

The next step is to specify what it means for a reordering to be admissible. Intuitively, only reorderings that preserve the search space of the query are considered admissible.

Definition 42 [Admissibility of Simple Reorderings] Let P be a program consisting of the sequence of clauses $c_1 \dots c_i \dots c_n$, where the clause c_i is

$$p : -\langle \theta_1, q_1 \rangle, \dots, \langle \theta_k \sqcup \{\lambda\}, q_k \rangle, \langle \theta_{k+1}, q_{k+1} \rangle, \dots, \langle \theta_m, q_m \rangle.$$

Let P' be a simple reordering of P , i.e. a program consisting of the sequence of clauses $c_1 \dots c'_i \dots c_n$, where the clause c'_i is

$$p : -\langle \theta_1, q_1 \rangle, \dots, \langle \theta_k, q_k \rangle, \langle \theta_{k+1} \sqcup \{\lambda\}, q_{k+1} \rangle, \dots, \langle \theta_m, q_m \rangle.$$

Let S be a query (i.e., a computation state) to program P . P' is a correct (admissible) simple reordering of P in the context of the query S if

$$\tau_P(S) \approx \tau_{P'}(S).$$

Definition 43 [Admissibility of Reorderings] P' is a correct (admissible) reordering of P if

1. P' is a correct simple reordering of P ; or

2. there is a program P'' such that P'' is a correct reordering of P and P' is a correct reordering of P'' .

Consider the programs P and P' in the definition of simple reordering. The program P can be modified in such a way that clause c_i becomes

$$p : - \langle \theta_1, q_1 \rangle, \dots, \langle \theta_k \sqcup \{\lambda\}, q_k \rangle, \langle \theta_{k+1} \sqcup \{\lambda\}, q_{k+1} \rangle, \dots, \langle \theta_m, q_m \rangle.$$

This program has the same search space as the original program since the second occurrence of λ is redundant. As a consequence, it is sufficient to show that this new program has the same search space as P' . Now observe that the new program and P' only differ by the fact that they add or do not add the constraint λ at program point k in clause c_i . With this observation in mind, it is convenient to reformulate the definition of the reordering problem and its admissibility by introducing a syntactic construct $?$ that can be applied to a constraint store and a constraint and whose semantics will be given by two different mapping functions, one corresponding to the original program (after addition of the redundant constraint) and the other corresponding to the reordered program. This simplifies the proofs significantly. The next definition captures this informal description.

Definition 44 [Revised Admissibility of Simple Reordering] Let P be a program consisting of the sequence of clauses $c_1 \dots c_i \dots c_n$, where the clause c_i is

$$p : - \langle \theta_1, q_1 \rangle, \dots, \langle \theta_k \sqcup \{\lambda\}, q_k \rangle, \langle \theta_{k+1}, q_{k+1} \rangle, \dots, \langle \theta_m, q_m \rangle.$$

Let P' be a simple reordering of P , i.e. the program consisting of the sequence of clauses $c_1 \dots c'_i \dots c_n$, where the clause c'_i is

$$p : - \langle \theta_1, q_1 \rangle, \dots, \langle \theta_k, q_k \rangle, \langle \theta_{k+1} \sqcup \{\lambda\}, q_{k+1} \rangle, \dots, \langle \theta_m, q_m \rangle.$$

Further, let R be a program consisting of the sequence of clauses $c_1 \dots c''_i \dots c_n$, where the clause c''_i is

$$p : - \langle \theta_1, q_1 \rangle, \dots, \langle \theta_k ? \{\lambda\}, q_k \rangle, \langle \theta_{k+1} \sqcup \{\lambda\}, q_{k+1} \rangle, \dots, \langle \theta_m, q_m \rangle.$$

We denote by $\tau_R^i(S)$ the computation tree with root node S where $\theta ? \{\lambda\} = \theta \sqcup \{\lambda\}$ and call it the “inclusive” execution of R . We also denote by $\tau_R^x(S)$ the computation tree with root node S where $\theta ? \{\lambda\} = \theta$ and call it the “exclusive” execution of R . Let S be a query to the program P . Then P' is an admissible simple reordering of P in the context of the query S if

$$\tau_R^i(S) \approx \tau_R^x(S).$$

Intuitively, the inclusive execution of a query corresponds to the execution in the original program, while the exclusive execution corresponds to the execution in the reordered program.

6.1.4 Failure-Free Reorderings

The characterization of admissible reorderings is not easily amenable to abstraction. This section presents failure-free reorderings which are based on a condition which guarantees admissibility and which is much more amenable to abstraction. The intuition is as follows for the reordering problem as stated in the previous definition. Informally speaking, the reordering is failure-free if we can prove that λ does not cause any failure of the goal $\langle \theta_k, q_k \rangle$ in the reordered program when executed with any input store of the goal $\langle \theta_k \sqcup \{\lambda\}, q_k \rangle$ occurring in the original program. We now formalize this notion and prove the correctness results.

We first introduce the concept of *failure freedom*. Essentially, a constraint is failure-free w.r.t. a (satisfiable) store if adding the constraint to the store does not cause the resulting store to become unsatisfiable. The concept can be extended to failure freedom over computation trees. A constraint is failure-free with respect to a computation tree if it is failure-free with respect to all constraint stores in that tree. More formally:

Definition 45 [Failure Freedom] The constraint λ is said to be failure-free with respect to the constraint store θ (denoted $\lambda \text{ FF } \theta$) if

$$\theta \text{ satisfiable} \Rightarrow \theta \sqcup \{\lambda\} \text{ satisfiable} .$$

Also, the constraint λ is said to be failure-free w.r.t. the computation tree T (denoted by $\lambda \text{ FF } T$) if for every constraint store θ in $\text{stores}(T)$, λ is failure-free w.r.t. θ .

We are now in position to define failure-free reorderings of programs.

Definition 46 [Failure-Free Reorderings] Let P be a program consisting of the sequence of clauses $c_1 \dots c_i \dots c_n$, where the clause c_i is

$$p : - \langle \theta_1, q_1 \rangle, \dots, \langle \theta_k \sqcup \{\lambda\}, q_k \rangle, \langle \theta_{k+1}, q_{k+1} \rangle, \dots, \langle \theta_m, q_m \rangle.$$

Let P' be a simple reordering of P , i.e. the program consisting of the sequence of clauses $c_1 \dots c'_i \dots c_n$, where the clause c'_i is

$$p : - \langle \theta_1, q_1 \rangle, \dots, \langle \theta_k, q_k \rangle, \langle \theta_{k+1} \sqcup \{\lambda\}, q_{k+1} \rangle, \dots, \langle \theta_m, q_m \rangle.$$

Further, let R be a program consisting of the sequence of clauses $c_1 \dots c_i'' \dots c_n$, where the clause c_i'' is

$$p : - \langle \theta_1, q_1 \rangle, \dots, \langle \theta_k ? \{\lambda\}, q_k \rangle, \langle \theta_{k+1} \sqcup \{\lambda\}, q_{k+1} \rangle, \dots, \langle \theta_m, q_m \rangle.$$

Denote by program point A the point in the clause c_i'' just before $\langle \theta_k ? \{\lambda\}, q_k \rangle$. Denote by Θ the set of all constraint stores that can occur as the accumulated constraint store at the program point A in the inclusive execution of program R for the query S . Then P' is a failure-free reordering of P for S if

$$\forall \theta \in \Theta : \lambda \text{ FF } \tau_R^\#(\langle \langle \theta_k ? \{\lambda\}, q_k \rangle \Diamond \theta \rangle).$$

We now prove that failure-free reorderings are admissible. The following lemma pertaining to failure freedom is a simple consequence of the definition.

Lemma 19 If λ is failure-free in a computation tree T , it is failure-free in any subtree thereof.

The next lemma gives a strong relation between failure freedom and similarity of trees.

Lemma 20 Let P be a program. Then for every satisfiable constraint store θ , every constraint λ and every body G ,

$$\lambda \text{ FF } \tau_P(\langle G \Diamond \theta \rangle) \Rightarrow \tau_P(\langle G \Diamond \theta \rangle) \approx \tau_P(\langle G \Diamond \theta \sqcup \{\lambda\} \rangle).$$

Sketch of Proof: The proof consists of proving that for any satisfiable constraint store θ , constraint λ , body G and finite depth d s.t. $\lambda \text{ FF } \tau_P(\langle G \Diamond \theta \rangle)$:

$$\tau_P(\langle G \Diamond \theta \rangle) \approx_d \tau_P(\langle G \Diamond \theta \sqcup \{\lambda\} \rangle).$$

The proof proceeds by induction on d . □

The next lemma enables us to strengthen a failure freedom hypothesis for programs that have a structure similar to our reorderings.

Lemma 21 Let θ be a satisfiable constraint store, P be a program, G and G' be a bodies, q be an atom, and θ_q be a store. Then

$$\lambda \text{ FF } \tau_P(\langle G \Diamond \theta \rangle) \Rightarrow \lambda \text{ FF } \tau_P(\langle G :: \langle \theta_q \sqcup \{\lambda\}, q \rangle :: G' \Diamond \theta \rangle).$$

Proof: For any store $\theta' \in \text{stores}(\tau_P(\langle G :: \langle \theta_q \sqcup \{\lambda\}, q \rangle :: G' \Diamond \theta \rangle))$, we have that either $\theta' \in \text{stores}(\tau_P(\langle G \Diamond \theta \rangle))$ or $\theta' = \{\lambda\} \sqcup \theta''$. The first case applies to the computation states that occur during the execution of the prefix G , while the second case applies to the computation states that occur during the execution after G . In the first case λ is failure-free w.r.t. θ' by the hypothesis. In the second case λ is failure-free w.r.t. θ' trivially. The desired result follows. \square

We now turn to the main result of this section which gives a sufficient condition for admissible reorderings.

Theorem 16 [Failure-Free Reorderings are Admissible] Let R be a program consisting of the sequence of clauses $c_1 \dots c_i \dots c_n$, where the clause c_i is

$$p : - \langle \theta_1, q_1 \rangle, \dots, \langle \theta_k ? \{\lambda\}, q_k \rangle, \langle \theta_{k+1} \sqcup \{\lambda\}, q_{k+1} \rangle, \dots, \langle \theta_m, q_m \rangle.$$

Denote by program point A the point in the clause c_i just before $\langle \theta_k ? \{\lambda\}, q_k \rangle$. Denote by Θ the set of all constraint stores that can occur as the accumulated constraint store at the program point A in the inclusive execution of program R for the query S . Let

$$\forall \theta \in \Theta : \lambda \text{ FF } \tau_R^{\mathcal{F}}(\langle \langle \theta_k ? \{\lambda\}, q_k \rangle \Diamond \theta \rangle).$$

Then

$$\tau_R^i(S) \approx \tau_R^{\mathcal{F}}(S).$$

Sketch of Proof: The proof considers a hybrid tree $\tau_R^h(S, d)$ which for any depth d corresponds to the inclusive execution of S upto depth d , and the exclusive execution thereafter. The definition of the tree is such that for any d , $\tau_R^h(S, d)$ is identical to $\tau_R^i(S)$ upto depth d . The proof consists of showing (by induction) that for any d ,

$$\tau_R^h(S, d) \approx \tau_R^{\mathcal{F}}(S).$$

It follows trivially that for any depth d :

$$\tau_R^i(S) \approx_d \tau_R^{\mathcal{F}}(S)$$

which is the desired result. \square

We would like to mention here that due to the technical details of the proof, it is necessary to compute the set of stores Θ in the inclusive execution of the query (i.e. the program before reordering); however the test for failure freedom of λ is performed w.r.t. an

exclusive computation tree (i.e. the program after reordering). This subtlety in the test for reordering was never mentioned previously in the literature, and it was discovered only because of formally defining the criterion for admissible reorderings and rigorously proving a sufficient condition for admissible reorderings.

Note that the admissibility of reorderings and the sufficient condition for admissible reorderings presented in this chapter have been given in terms of a modified operational semantics, that differs from the standard semantics because at each computation step, it adds all the adjacent constraints simultaneously to the accumulated constraint store rather than add them individually in multiple computation steps. However it is not a difficult matter to model the standard operational semantics and search space of CLP programs in terms of the operational semantics and search space given in this chapter. All that needs to be done is to force all constraints to be considered as singleton stores and to impose the restriction that the search space cannot be modified even when reordering among constraints.

Capturing the ideas of the previous paragraph, the following procedure may be used to decide whether λ_1 can be reordered to the point just after q (where q may be a predicate or a constraint) in the clause

$$p : - \dots, \lambda_1, \dots, q, \dots$$

while preserving the search space of the standard operational semantics:

1. Replace every constraint λ of the program except λ_1 by a new predicate symbol p_λ ;
2. Add the clauses $p_\lambda : -\lambda.$ to the program;
3. Rewrite the program in terms of the modified syntax;
4. Decide if moving λ_1 after q (a sequence of simple reordering steps) is admissible as per Theorem 16.

As a special case of the above, the following procedure may be used to decide whether λ_1 can be reordered after the constraint λ in the clause

$$p : - \dots, \lambda_1, \lambda, \dots$$

while preserving the search space of the standard operational semantics. Let Θ be the set of all constraint stores that can occur at the point in the clause just before λ_1 in the

execution of the original program. Then it is admissible to move λ_1 after λ if

$$\forall \theta \in \Theta : \lambda_1 \text{ FF } \theta \wedge \lambda_1 \text{ FF } \theta \sqcup \{\lambda\}.$$

6.1.5 Abstract Test for Reordering

The aim of this section is to show how the main result of the previous section (failure-free reorderings are admissible) may be abstracted safely to give an abstract test for reordering to be used in an optimizing compiler. The key idea is to recognize that failure freedom of a constraint w.r.t. a store is exactly the same as the conditional satisfiability operation introduced in Section 3.4.3, i.e.

$$\lambda \text{ FF } \theta \Leftrightarrow \text{Cis_cond_sat}(\theta, \{\lambda\}).$$

This suggests that the domains LSign and LInt can be used to obtain an abstract test for deciding admissible reorderings.

Corollary 2 Let R be a program consisting of the sequence of clauses $c_1 \dots c_i \dots c_n$, where the clause c_i is

$$p : - \langle \theta_1, q_1 \rangle, \dots, \langle \theta_k ? \{\lambda\}, q_k \rangle, \langle \theta_{k+1} \sqcup \{\lambda\}, q_{k+1} \rangle, \dots, \langle \theta_m, q_m \rangle.$$

Denote by program point A the point in the clause c_i just before $\langle \theta_k ? \{\lambda\}, q_k \rangle$. Denote the set of all constraint stores that can occur as the accumulated constraint store at the program point A in the inclusive execution of program R for the query S by Θ and denote $\text{stores}(\tau_R^\pi(\langle \langle \theta_k ? \{\lambda\}, q_k \rangle \Diamond \emptyset))$ by Θ' . Let α and α' be abstract descriptions and γ be an abstract constraint with multiplicity s.t.

$$\Theta_{/\text{var}(c_i)} \subseteq Cc(\alpha), \Theta'_{/\text{var}(c_i)} \subseteq Cc(\alpha'), \lambda \in Cc(\gamma),$$

and

$$\text{Ais_cond_sat}(\alpha \uplus \alpha', \{\{\gamma\}\}).$$

Then

$$\tau_R^i(S) \approx \tau_R^\pi(S).$$

Proof: By the consistency of the \uplus operator and the conditional satisfiability operation, we have that

$$\forall \theta \in \Theta_{/\text{var}(c_i)} \forall \theta' \in \Theta'_{/\text{var}(c_i)} : \text{Cis_cond_sat}(\theta \sqcup \theta', \{\lambda\}),$$

Since the conditional satisfiability of λ w.r.t. $\theta \sqcup \theta'$ depends only on the projection of $\theta \sqcup \theta'$ on the variables of λ , we have that

$$\forall \theta \in \Theta \forall \theta' \in \Theta' : \text{Cis_cond_sat}(\theta \sqcup \theta', \{\lambda\}),$$

Using the commutativity of constraint addition in the domain of real linear constraints, this can be rewritten as

$$\forall \theta \in \Theta \forall \theta' \in \text{stores}(\tau_R^\pi(\langle \langle \theta_k ? \{\lambda\}, q_k \rangle \Diamond \theta \rangle)) : \text{Cis_cond_sat}(\theta', \{\lambda\}).$$

By using the definition of failure freedom for trees and the fact that

$$\lambda \text{ FF } \theta \Leftrightarrow \text{Cis_cond_sat}(\theta, \{\lambda\}).$$

it follows that

$$\forall \theta \in \Theta : \lambda \text{ FF } \tau_R^\pi(\langle \langle \theta_k ? \{\lambda\}, q_k \rangle \Diamond \theta \rangle).$$

The desired result follows by Theorem 16. \square

In order to determine the admissibility of a reordering in terms of the standard operational semantics, the abstract test should be used to approximate the fourth step of the four step procedure in the previous subsection. In that case, α can be computed given the input description of p for the query S in the original program, and the goal independent output descriptions of q_1, \dots, q_{k-1} . α' is just the goal independent intermediate description of q_k in the reordered program (as θ_k is empty).

As a special case of the above, the following procedure may be used to decide whether λ_1 can be reordered after the constraint λ in the clause

$$p : - \dots, \lambda_1, \lambda, \dots$$

while preserving the search space of the standard operational semantics. Let Θ be the set of all constraint stores that can occur at the point in the clause just before λ_1 in the execution of the original program. Let α be an abstract description s.t. $\Theta \subseteq Cc(\alpha)$; γ_1 and γ be abstract constraints s.t. $\lambda_1 \in Cc(\gamma_1)$ and $\lambda \in Cc(\gamma)$. Then it is admissible to move λ_1 after λ if

$$\text{Ais_cond_sat}(\alpha, \{\{\gamma_1\}\}) \wedge \text{Ais_cond_sat}(\alpha \uplus \gamma, \{\{\gamma_1\}\}).$$

α may be computed using the abstract interpretation framework described in Chapter 2 from the input description of p for the query and the goal independent output descriptions of the intervening predicate calls before λ in the clause.

6.1.6 Complete Example

Consider the normalized mortgage program

$\text{mg}(P, T, R, B) :- T = 0, B = P.$

$\text{mg}(P, T, R, B) :- T > 0, P \geq 0, P1 = P * 1.01 - R, T1 = T - 1, \text{mg}(P1, T1, R, B).$

and assume that the compiler tries to move constraint $T > 0$ after the recursive call.

This actually consists of a sequence of reordering steps, the first of which attempts to move $T > 0$ after the constraint $P \geq 0$. The proof obligation (abstract test) consists of checking whether $T > 0$ is failure-free w.r.t. the input store of mg in the original program (α_{in}) which is given by

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

and also if $T > 0$ is failure-free w.r.t. $\alpha_{in} \uplus \langle 0 \leq P, \text{One} \rangle$. The method used to check failure freedom is of course the conditional satisfiability in the domain LSign . The abstract test consists of checking whether

$$\text{Ais_cond_sat}(\alpha_{in}, \{\{\langle 0 < \oplus T, \text{One} \rangle\}\}) \wedge \text{Ais_cond_sat}(\alpha_{in} \uplus \langle 0 \leq P, \text{One} \rangle, \{\{\langle 0 < \oplus T, \text{One} \rangle\}\})$$

is true and it succeeds.

The next two reordering steps involve moving $T > 0$ after the constraints $P1 = P * 1.01 - R$ and $T1 = T - 1$ and similar proof obligations can be carried out by checking the failure freedom of $T > 0$ w.r.t. $\alpha_{in} \uplus \langle 0 \leq \oplus P, \text{One} \rangle$, $\alpha_{in} \uplus \langle 0 \leq \oplus P, \text{One} \rangle \uplus \langle 0 = \oplus P + \ominus R + \oplus P1, \text{One} \rangle$ and $\alpha_{in} \uplus \langle 0 \leq \oplus P, \text{One} \rangle \uplus \langle 0 = \oplus P + \ominus R + \oplus P1, \text{One} \rangle \uplus \langle \oplus = \oplus T + \ominus T1, \text{One} \rangle$.

The final reordering step consists of moving $T > 0$ after the recursive call to mg . It is now necessary to carry out the proof obligation of Theorem 16 by means of the abstract test in Section 6.1.5. The store α in the abstract test is $\alpha_{in} \uplus \langle 0 \leq \oplus P, \text{One} \rangle \uplus \langle 0 =$

$\oplus P + \ominus R + \oplus P1, \text{One}\rangle \sqcup \langle \oplus = \oplus T + \ominus T1, \text{One}\rangle$. The store α' is

```
{
  { }
  {⟨0 = ⊕T1, One⟩},
  {⟨0 ≤ ⊕P1, One⟩}
  {⟨0 ≤ ⊕P1, One⟩, ⟨⊕ = ⊕T1, One⟩}
  {⟨0 ≤ ⊕P1, One⟩, ⟨⊕ = ⊕T1, One⟩, ⟨0 = ⊕P1 + ⊖R + ⊖B, One⟩}
  {⟨0 ≤ ⊕P1, One⟩, ⟨⊕ = ⊕T1, One⟩, ⟨0 = ⊕P1 + ⊖R + ⊖B, One⟩, ⟨0 < ⊕T1, One⟩}
  {⟨0 ≤ ⊕P1, One⟩, ⟨⊕ = ⊕T1, One⟩, ⟨0 ≤ ⊕P1 + ⊖R, One⟩, ⟨0 ≤ ⊕P1 + ⊖R, Any⟩}
  {⟨0 ≤ ⊕P1, One⟩, ⟨⊕ = ⊕T1, One⟩, ⟨0 = ⊕P1 + ⊖R + ⊖B, One⟩,
    ⟨0 ≤ ⊕P1 + ⊖R, One⟩, ⟨0 ≤ ⊕P1 + ⊖R, Any⟩}
  {⟨0 ≤ ⊕P1, One⟩, ⟨0 ≤ ⊕P1 + ⊖R, One⟩, ⟨0 ≤ ⊕P1 + ⊖R, Any⟩}
  {⟨0 = ⊕T1, One⟩, ⟨0 = ⊕P1 + ⊖B, One⟩},
  {⟨0 < ⊕T1, One⟩, ⟨⊕ = ⊕T1, One⟩, ⟨0 ≤ ⊕P1, One⟩,
    ⟨0 = ⊕P1 + ⊖R + ⊖B, One⟩, ⟨0 ≤ ⊕R + ⊖B, One⟩, ⟨0 ≤ ⊕R + ⊖B, Any⟩}
}
```

The abstract test consists of verifying that the constraint $T > 0$ is failure-free w.r.t. the complicated abstract description $\alpha \sqcup \alpha'$ consisting of 33 abstract stores. This is verified by checking the truth of $\text{Ais_cond_sat}(\alpha \sqcup \alpha', \{\{\langle 0 < \oplus T \rangle\}\})$, and hence the reordering is admissible.

6.2 Removal Optimization

The removal optimization phase receives the reordered program and performs the LSign or LInt analysis on the reordered program. The optimizer then systematically considers each constraint in each clause for constraint removal. To understand when a constraint can be removed from the program, consider a clause

$$p :- g_1, \dots, g_i, \lambda, g_{i+1}, \dots$$

and assume that the compiler is interested in determining whether the constraint λ can be removed from the body of the clause. A sufficient condition for removing λ is that for all accumulated constraint stores θ that can occur at the program point before λ during the execution of the program for a given query, $\text{Cis_redundant}(\{\lambda\}, \theta)$ is true. The abstract interpretation of the program enables us to obtain a LSign or LInt representation of

all such accumulated constraint stores, call this abstract description α . The condition can then be verified in a conservative manner by checking if $\text{Ais_redundant}(\{\{\gamma\}\}, \alpha)$ is true, where γ is an abstraction of λ . Consider the running example again. The program at this stage is as follows:

$\text{mg}(P, T, R, B) \text{ :- } T = 0, B = P.$

$\text{mg}(P, T, R, B) \text{ :- } P \geq 0, P1 = P * 1.01 - R, \text{mg}(P1, T1, R, B), T = T1 + 1, T > 0.$

The LSign analysis produces the description $\alpha =$

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

for the program point just before the constraint $T > 0$. The first abstract multistore represents the input to the clause, the second abstract multistore represents the constraints in the clause before $T > 0$ and the third abstract multistore represents the output of the recursive call. This is a fairly complex abstract description containing nine abstract stores. The optimizer verifies that $\text{Ais_redundant}(\{\{\langle 0 < \oplus T, \text{One} \rangle\}\}, \alpha)$ is true, and so the constraint is redundant and can be removed from the program.

6.3 Refinement Optimization

After constraint removal, the refinement phase considers all constraints in all clauses and specializes them whenever possible. It performs both a Prop analysis and an LSign or LInt analysis on the program obtained after reordering and removal. The specialization

of inequalities only uses the results of Prop and produces a test whenever all variables are fixed. The specialization of equations also uses the results of LSign to determine unconstrained variables. An equation is specialized into a test if all its variables are fixed and into an assignment if all but one of its variables are fixed, and the non-fixed variable is free. Consider the running example during this phase:

```
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.
```

Given the top level input pattern, the abstract description for all the constraint stores that can occur before the constraint $P \geq 0$ is just the input description for all calls to mg and it is given by $f = P \wedge R$ in the Prop domain and $\alpha =$

$$\{ \begin{aligned} &\langle T = \oplus P, \text{One} \rangle, \langle T = \oplus R, \text{One} \rangle, \\ &\langle T = \oplus R, \text{One} \rangle, \langle T = \oplus P + \oplus R, \text{One} \rangle, \\ &\langle T = \oplus R, \text{One} \rangle, \langle T = \oplus P + \oplus R, \text{One} \rangle, \langle T \leq \ominus R, \text{One} \rangle, \langle T \leq \ominus R, \text{Any} \rangle \end{aligned} }$$

in the LSign domain. In the second clause, the first inequality is transformed into a test, since $\text{Ais_fixed}(f, P)$ i.e. P is fixed in the input description of mg. The abstract description before the second constraint is the same in the Prop domain and is $\alpha' = \alpha \uplus \langle 0 \leq \oplus P, \text{One} \rangle$ in the LSign domain. Since $\text{Ais_fixed}(f, R)$ and $\text{Ais_free}(\alpha', P1)$ i.e. R is also fixed and $P1$ is unconstrained, the second constraint can be transformed into an assignment. Similar reasoning can be applied for the remaining constraints to obtain the refined program:

```
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.
```

6.4 Integration

Finally, we would like to mention that the compiler is organized in three passes for optimization, viz. reordering, removal and refinement, as indicated in Figure 6.2. There is also a normalization phase before the optimization phases. Note that the removal and refinement passes may be interchanged with little effect on the overall optimization. However it is important to reorder first because the reordering is designed to increase

the amount of refinement and redundancy removal. The program structure is passed between the phases through intermediate data structures¹. An abstract interpretation system based on GAIA is adapted to CLP for the purposes of computing the various abstract descriptions needed by the optimizer.

In the reordering phase, the abstract interpretation system is used to recompute the abstract descriptions after each reordering step. This is because the abstract descriptions of interest (input and intermediate descriptions) may change when the program is reordered. Presently, the recomputation is not incremental, i.e. the descriptions are recomputed from scratch after each simple reordering step. There is no need to recompute descriptions in the removal and refinement phases.

We have not discussed how potential target sites to reorder constraints are determined. Typically it is sought to move constraints to points in the program where they become tests or assignments or where they become redundant. The analyses for refinement (Prop and LSign) and redundancy (LSign) are thus used to guide the reordering stage to attempt profitable reorderings. The present compiler does not use any sophisticated strategy to select among reorderings. Any reordering that could move a constraint to a point where it becomes a test or assignment is considered for analysis.

¹An earlier version of the compiler used intermediate files containing the transformed program.

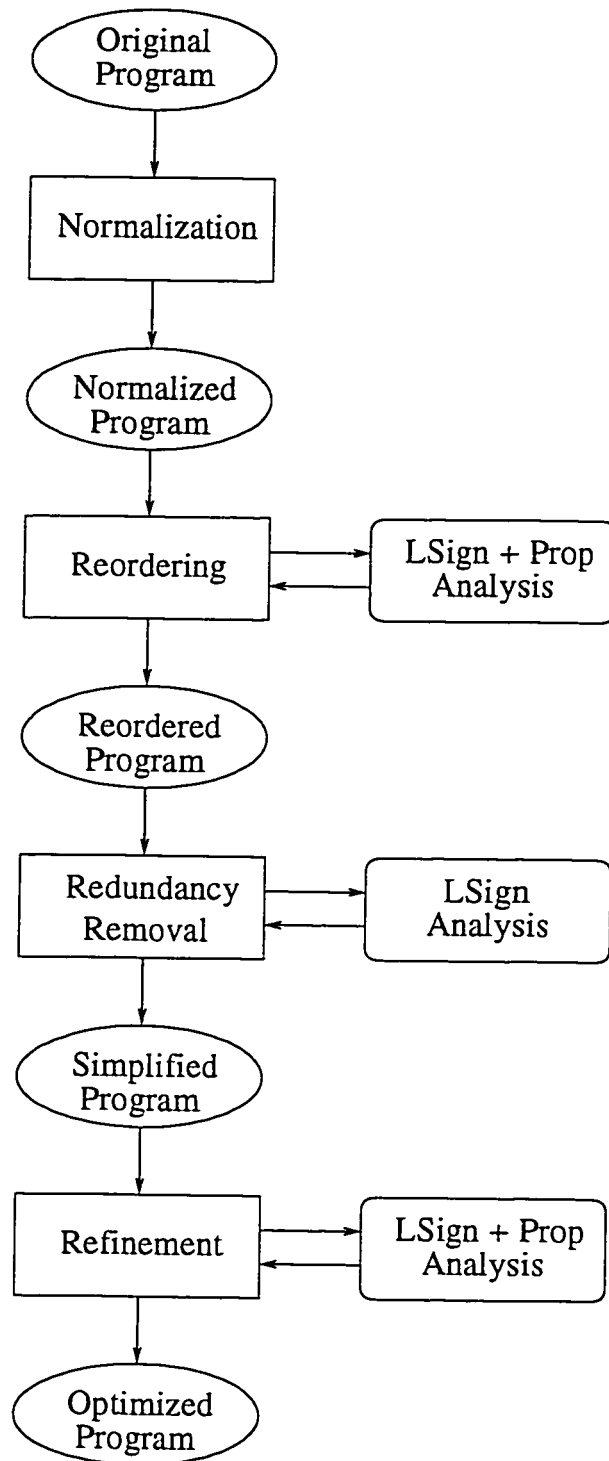


Figure 6.2: Organization of the Compiler and Optimizer

Chapter 7

Experimental Results

The purpose of this chapter is to evaluate the effect of the optimizations on a number of typical benchmarks. The chapter quantifies the benefit of applying the optimizations by comparing the execution time for optimized and unoptimized versions of the benchmarks. It also gives the optimization times for the benchmarks and compares the optimization times of the LSign analysis and LInt analysis. The chapter is organized as follows. Section 7.1 presents the benchmarks used in the experimental results. The execution time for optimized and unoptimized versions of the benchmarks is compared in Section 7.2. The optimization time for LSign and LInt analyses is presented in Section 7.3.

7.1 The Benchmarks

Our benchmarks are relatively small, since most large $\text{CLP}(\mathcal{R}_{Lin})$ programs use Prolog terms in addition to linear constraints, and our analyzer is not able to handle structures at this stage. Nevertheless, the benchmarks indicate clearly the potential of the optimizations. 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^N , the sum of the first N integers, and the N^{th} Fibonacci number, respectively. Once again, these benchmarks are used in various modes such as computing the sum of the first N integers, determining N

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

Table 7.1: Test Programs: Description of Usage in Various Modes.

such that a given number F is the N^{th} Fibonacci number, generating all pairs of numbers $\langle N, 2^N \rangle$ for $0 \leq N \leq 25$ 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 thesis. 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 [39] 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 7.1 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. In other words, the constraint solver can deduce some of the variables to be fixed, and using their fixed values instead of their symbols makes the constraints linear.

The benchmarks used for testing the analyses with LSign and LInt are similar. The

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 4000 8000		520	130	4.00	REF,REO	✓
		1660	210	7.90	REF,REO	✓
		4490	290	15.48	REF,REO	✓
Ar. Mean				5.68		

Table 7.2: Comparison of Running Times: Optimized vs. Unoptimized.

only difference is that the version of a benchmark used with LSign typically has a base case of zero, while the version of the same benchmark used with LInt has a non-zero base case.

7.2 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 runtime system uses infinite precision integers to guarantee numerical stability. Table 7.2 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 \checkmark indicates that 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. It is interesting to see which optimizations had a greater effect on the speedups. The average speedup when only refinements were performed was just 1.20. This is because when only refinements are performed, even the unoptimized program does not invoke the constraint solver for the constraints that are refined. The only saving comes from the specialization into tests and assignments at compile time, rather than at runtime. This is because the system is organized such that tests and assignments discovered at runtime are executed in the interface rather than being passed to the constraint solver. When the optimizations do not result in bypassing the constraint solver entirely, the average speedup observed was 2.23. This speedup is about double that obtained for refinements only. The maximum benefits were observed when the constraint solver was bypassed entirely. In that case, the average speedup was 8.39. This is because the original program performed costly constraint solving, while the optimized program only executed tests and assignments (handled in the interface and engine). Note that there is a close correlation between reordering and bypassing the solver entirely. Each of the examples where the solver is bypassed entirely was reordered by the optimizer (though the reverse is not true). Reordering moves the constraints to a point in the program where they may be removed or specialized into tests or assignments, thereby bypassing the solver.

The speedups on *Triangular* illustrate that the optimizations can produce more than constant factors of improvement. In *Triangular*, the speedup depends on the size of the problem, i.e. the number of equations being solved simultaneously. Reordering and refinement make it possible to bypass calls to the constraint solver completely, avoiding the costly Gaussian elimination in the solver which takes place in the original program. While the original program solves a set of constraints which takes time quadratic in the number of variables, the optimized program executes a sequence of assignments which

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 (Linear)	fffu	120	1420	20	20	1460
	fufu	100	3060	30	10	3100
Mortgage (Nonlinear)	fffu	160	3350	50	40	3440
	fuffu	240	9480	50	30	9560
Ode-Euler	fffu	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 7.3: Optimization Times: LSign.

takes time linear in the number of variables.

Mortgage (linear and nonlinear versions) also illustrates a side-effect of reordering that we did not expect. As mentioned previously, the system uses infinite-precision numbers but it makes an effort (especially in the interface between the engine and the solver) to keep numbers in standard integer technology as long as possible. When a program is reordered, it is possible that it can be run without infinite precision numbers at all. When this happens, the speedups are magnified and this explains the discrepancy between the speedups of Mortgage with a value 200 and with a value 100 for the principal.

The above results are presented for the optimizing compiler using the LSign analysis. The results are very similar using the LInt analysis and the modified benchmarks. The main point of interest is that the modified benchmarks typically cannot be optimized by the LSign analysis, however they are successfully optimized by the LInt analysis.

7.3 Optimization Time

In order to get an idea whether the optimizations are practical, it is useful to quantify the time taken to perform reordering, refinement and removal in the optimizing compiler.

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 (Linear)	fffu	14410	1460	9.87
	fufu	42420	3100	13.68
Mortgage (Nonlinear)	fffu	25070	3440	7.29
	fuffu	62180	9560	6.50
Ode-Euler	fffu	11110	1730	6.42
	fufff	11070	1700	6.51
	fuffu	7200	1180	6.10
Triangular, 2000 4000 8000		1350	390	3.46
		1510	520	2.90
		3920	2450	1.60
Ar. Mean				7.39

Table 7.4: Comparison of Optimization Times: LInt vs. LSign.

Table 7.3 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. The table indicates that the optimization times are reasonable for our benchmark programs, but that avenues of making the optimizer fast is an open area of research. For example the techniques of [19] should be useful here to obtain a faster, incremental implementation.

In order to experimentally quantify the penalty paid for using intervals instead of signs, it is also useful to compare the optimization times when LInt is used in the analyzer instead of LSign. Table 7.4 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 indicating that the LInt analysis is practical, and does not impose too much overhead for the additional accuracy.

Chapter 8

Related Work

In this chapter, we discuss the relation of our thesis with a number of different areas of research. The chapter is organized as follows. Section 8.1 discusses the relation of the work presented in this thesis with the research on optimization of the language $\text{CLP}(\mathcal{R})$. The previous work on abstract interpretation of CLP languages is discussed in Section 8.2. The relation of the thesis with optimization of Prolog is covered in Section 8.3. The chapter concludes by examining the relation of the $\text{CLP}(\mathcal{R}_{Lin})$ optimizations with imperative language optimizations in Section 8.4.

8.1 Optimization of $\text{CLP}(\mathcal{R})$

Our research is closely related to a number of papers published on the optimization of $\text{CLP}(\mathcal{R})$ (eg. [36, 26, 37, 24, 39, 31, 23]). However, few experimental results have appeared to quantify the possible benefits in a practical optimizing compiler. The only exceptions that we are aware of is the system described in [26] which performs three optimizations on $\text{CLP}(\mathcal{R})$ programs viz. “solver bypass”, “dead variable elimination” and “no fail constraint detection” and the system described in [27] which performs the above optimizations, as well as reordering and an optimization called “future redundancy”. However little information is available as to how the reordering is automated or how the correctness of the reordering optimization is guaranteed in the system of [27]. It is just mentioned that a constraint that contains a “new” variable can be reordered across any number of intervening goals that do not contain that variable. This is actually just a special case of the conditional satisfiability checked in our compiler (as the projection of any intermediate store on the “new” variable would be the empty set). Our compiler’s

reordering criterion is much more sophisticated and can allow reorderings to take place even when all the variables of the constraint under consideration occur in the intervening goals. As we show in this thesis, reordering is probably the most fundamental optimization, enabling the other optimizations to be applied more effectively. Moreover, there are several subtle issues in guaranteeing that the reordered program has the same search space as the original program, and these deserve to be addressed in rigorous detail. While reordering constraints was first proposed by Marriott and Stuckey [36, 37], this thesis is the first to formalize the notion of admissible reorderings and give an abstract test for the same.

Our optimizations are most closely related to the refinement, removal and reordering (3R's) proposed in [36]. Unlike our approach, the 3R's methodology transforms the original monotonic program (constraint addition only) into a non-monotonic program (constraint addition and removal). The refinement proposed in that paper is quite different from the refinement that we propose, and involves adding additional constraints to the source program, so as to guide the execution away from unprofitable choices. The redundancy elimination proposed in that paper is much more general, and is related to the future redundancy optimization of [24]. It consists of adding explicit constraint removal instructions to the program, which means that constraints can be added and then removed at a future program point when they have become redundant. Finally, reordering is also more general because in addition to moving constraint addition later (as in our compiler), the methodology also envisions moving constraint removal earlier. The paper also presents an outline of the analyses that would enable implementation of the 3R's methodology and an idea of the speedups obtainable, by optimizing some programs by hand.

There are number of other optimizations of $\text{CLP}(\mathcal{R})$ programs proposed in various papers. Optimizations such as “mutual exclusion” (converting multiple clauses for a predicate into “if then else” statements) and “code motion” (moving recursion invariant expressions out of recursive clauses, similar to loop invariant optimization in imperative languages) are proposed in [24]. More specialized optimizations such as linear threading, linear gather and mixed nonlinear gather, that work only on some classes of $\text{CLP}(\mathcal{R})$ programs are presented in [39].

8.2 Abstract Interpretation of CLP

As mentioned previously, the optimization of CLP programs is achieved by the method of abstract interpretation. Abstract interpretation of CLP is closely related to the abstract interpretation of logic programs and borrows many of the theoretical aspects such as work on frameworks [2, 5, 4, 32, 33, 38] and algorithms [5, 7, 29, 25, 41]. Much of the previous work on abstract interpretation of CLP has focused on theoretical aspects [17, 35] and defining various abstract domains [13, 14, 37]. Little work has been done on actually integrating domains into optimizing compilers and evaluating the benefits (except for the systems of [26, 27]). While this thesis contributes to the definition of the abstract domains LSign and LInt and the theory underlying them, it is also an attempt to demonstrate the use of abstract interpretation of CLP in a practical optimizing compiler and to quantify the benefits thereof.

8.3 Optimization of Prolog

The work on optimization of CLP languages can be seen as a natural extension of similar work on Prolog. In particular, the attempt to reduce the constraint solving time through refinement and removal can be compared with the attempt to reduce the time spent in unification. Papers by Van Roy [53] and Taylor [45] give a good idea of the speedups that can be obtained for Prolog programs through sophisticated global analyses. Debray [12] proposes a number of loop optimizations for Prolog that are the equivalent of classic loop optimizations in imperative languages. The paper uses “fold/unfold” transformations to remove recursion, fuse loops and move code out of loops. The $\text{CLP}(\mathcal{R}_{Lin})$ optimizations in our compiler are orthogonal to the above mentioned Prolog optimizations. In fact, as demonstrated earlier in the thesis, a $\text{CLP}(\mathcal{R}_{Lin})$ program may be transformed by our optimizations into a Prolog program (enhanced with a rational arithmetic component). All the traditional Prolog optimizations may then be applied to this program.

8.4 Optimization of Imperative Languages

Finally, we would like to add some comments about the relation of our work with the traditional compiler optimizations for imperative languages [1]. *Dead code elimination* in imperative languages is the counterpart to constraint removal in CLP languages. The interesting parallel is that just as dead code in imperative programs is likely to appear

as the result of previous transformations, so is removal of redundant constraints likely to occur due to previous reordering. *Code motion* in imperative languages can be compared with constraint reordering in CLP languages. While code motion typically seeks to move a loop invariant computation before the loop, constraint reordering typically seeks to move a constraint that does not influence a predicate call after the predicate call. The effect of code motion is to compute an expression just once, while the effect of constraint reordering is to execute the constraint at a point where it may be specialized to a cheaper operation or even removed altogether. However, we would like to point out that the analysis for reordering constraints is much more involved when compared to the analysis for moving loop invariant computation out of the loop. While the latter primarily involves reasoning about the local properties of the loop, the analysis for constraint reordering requires reasoning about the search space of the program which may be complicated due to nondeterministic definition of predicates. Finally, *strength reduction* in imperative languages may be compared to refinement in CLP languages. They both involve replacing more complicated expression evaluation by simpler forms, through static analysis.

Chapter 9

Conclusion

In this thesis, we have considered the problem of designing and implementing an optimizing compiler for the constraint logic programming language $\text{CLP}(\mathcal{R}_{Lin})$, performing the optimizations of reordering constraints, refining constraints into tests and assignments and removing redundant constraints. In this concluding chapter, we review the contributions of this thesis in Section 9.1 and discuss the open issues that remain in Section 9.2.

9.1 Contributions

The contributions of this thesis range from the theoretical, i.e. designing the abstract domains LSign and LInt and formalizing the reordering optimization, to the practical, i.e. implementing an optimizing $\text{CLP}(\mathcal{R}_{Lin})$ compiler.

The thesis contributes to the theory underlying the abstract domain LSign [37, 43]. Unfortunately, the original paper on LSign by Marriott and Stuckey [37] has a number of theoretical drawbacks. In particular, the ordering of abstract constraint stores given in [37] does not capture the intended meaning and makes it impossible to prove the consistency of the abstract operations of LSign . This thesis reconsiders the domain LSign , and corrects and completes the results of [37]. The thesis reports the first implementation of the domain LSign .

The thesis also proposes the domain LInt which is a generalization of LSign , abstracting coefficients by intervals instead of signs. This enables a more accurate analysis of programs in many cases. The main technical difficulty is that LInt is an infinite

domain unlike LSign, and therefore requires the definition of a widening operator to guarantee the termination of analyses. The implementation of LInt indicates that the domain can provide a practical analysis of programs and does not pay too much penalty in time of analysis as compared to LSign.

The thesis is the first work to formally specify what it means for a reordering optimization on CLP programs to be correct and it is also the first to present (with proof of correctness), a sufficient condition for performing correct reordering optimizations of $\text{CLP}(\mathcal{R}_{Lin})$ programs. The correctness criterion essentially guarantees that for any finite execution of the original program, the reordered program must mimic it. The thesis also shows how the correctness criterion may be reduced to a satisfiability problem on constraint stores which can then be answered (in a conservative fashion) using the domains LSign or LInt.

The primary aim of this thesis is to implement an optimizing compiler for $\text{CLP}(\mathcal{R}_{Lin})$, and we have succeeded in doing so. The experimental results obtained indicate the promise of our approach. The thesis presents the first provably correct implementation of reordering as well as the first implementation of constraint removal for $\text{CLP}(\mathcal{R}_{Lin})$ programs. Along with the implementation of constraint refinement, this produces the first $\text{CLP}(\mathcal{R}_{Lin})$ compiler integrating all these source to source transformations. The experimental results show that sophisticated static analyses and source to source transformations can produce dramatic speedups for $\text{CLP}(\mathcal{R}_{Lin})$ programs, indicating the promise of our approach. The speedups can increase as the size of the inputs increases. Also, reordering is seen to be the most powerful optimization, producing the most dramatic speedups. This is because reordering a program often leads to a complete bypass of the constraint solver. In that case, the unoptimized program performs costly constraint solving, while the optimized program performs only tests and assignments which are cheaper to implement.

9.2 Open Issues

This thesis leaves a number of minor and major problems in the area unsolved. We now mention these open issues.

The primary limitation of the solution to reordering presented in this thesis is the restriction to $\text{CLP}(\mathcal{R}_{Lin})$ programs without functors. CLP programs contain Prolog terms (structures) in addition to arithmetic constraints, and arithmetic expressions may

occur inside terms. This requires any abstract domain dealing with the properties of the variables in arithmetic constraints to also handle the cases where arithmetic expressions occur inside structures. It is not clear how to extend the domains `LSign` and `LInt` so as to handle structural information as well. This is a drawback in analyzing real-life CLP programs as they usually combine structures and arithmetic expressions. It is necessary to keep track of structural information in order to extend the scope of this work. An example that motivates the need for keeping structural information in the abstract domains follows. Consider a variant of the mortgage example where the monthly repayment may vary and where the final balance is always zero. The repayment parameter is replaced by a list of repayments and the parameter for the number of installments can be dropped as it is equal to the length of the list of repayments. This program is given below:

```
mglist(0,[ ]).
mglist(P,[R|Tail]) :- P ≥ 0, R ≥ 0, P1 = P*1.01 - R, mglist(P1,Tail).
```

When `mglist` is called such that the second argument (list of repayments) is fixed and the first argument (principal P) is unconstrained, it should be possible to reorder the program so as to produce

```
mglist(0,[ ]).
mglist(P,[R|Tail]) :- R ≥ 0, mglist(P1,Tail), P = (P1 + R) / 1.01, P ≥ 0.
```

However, the current `LSign` analysis is unable to handle structures (or lists), and therefore is unable to perform the above reordering. Unfortunately, it is not obvious or trivial to instantiate the generic pattern domain $\text{Pat}(\mathcal{R})$ [9] with the domains handling linear constraints. This is because $\text{Pat}(\mathcal{R})$ may lose information about functors. For example if either $R = f(T)$ or $R = g(T)$ is true (this could be imposed by two different clauses of a predicate, for example) then the generic pattern domain would lose the information that R is bound to a functor. In that case a future constraint such as $R = f(B)$ would fail to add the abstraction of the possible constraint $T = B$ to the abstract description, thereby violating the definition of the concretization for `LSign` and `LInt`. Some investigation is required to design a domain that can wrap `LSign` and `LInt` with structural information or to design an extended `LSign` domain (including for example some sharing information) that can be successfully instantiated into the $\text{Pat}(\mathcal{R})$ framework.

The `LSign` and `LInt` domains are not very accurate for the purposes of redundancy removal. Consider for example the constraint store $\{R + P \leq 1, P + B \leq 2, B + R \leq 5\}$.

The domains cannot deduce that the constraint $R + P + B \leq 4$ is redundant in the context of this store. This because the redundancy algorithms for the domains merely use the equations of the store to simplify the constraint and then check for trivial satisfiability. The domains cannot deduce redundancy caused by complex interactions of constraints. Utilizing a more sophisticated domain such as the domain Hull of [11] would improve the accuracy of redundancy removal.

The addition of structures and improved redundancy removal would enable the analysis of more real-life examples, including larger programs. While the present examples are not trivial, they are all short programs. Future work can focus on studying the behaviour and speedup obtained by the optimizations for large programs, to verify that they scale well.

Finally, not much attention has been paid to making the optimizing compiler efficient or to optimizing the abstract operations of the domains. In particular, the present compiler is naive with respect to the recomputation of abstract stores after each simple reordering step. Basically the input and intermediate description of every predicate that can be affected by a reordering is recomputed from scratch. Incremental techniques to only recompute relevant abstract stores of an abstract description need to be devised and the techniques of [19] should be helpful here to obtain a fast implementation. Also the present compiler performs multiple passes over the input program looking for more opportunities for reordering, as long as the previous pass was able to reorder something. Strategies could be devised to minimize the number of passes over the program. The abstract operations of the LSign and LInt domains can be made more efficient through the use of caching as has been done in implementations of other domains such as Prop. During the computation of a fixpoint for abstract descriptions, the same operations may get repeated several times on a given abstract store. Caching the result of previous operations could improve the performance.

Appendix A

Proofs of Results

A.1 Ordering

Lemma 22 [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 $\forall \gamma_1, \gamma_2 \in \beta_1 : \gamma_1 \neq \gamma_2 \Rightarrow f(\gamma_1) \neq f(\gamma_2) \vee f(\gamma_1) \in \text{Indef}(\beta_2)$ and $\forall \gamma_1, \gamma_2 \in \beta_2 : \gamma_1 \neq \gamma_2 \Rightarrow g(\gamma_1) \neq g(\gamma_2) \vee g(\gamma_1) \in \text{Indef}(\beta_3)$. By combination of the two statements and $f(\gamma_1) \sqsubseteq (g \circ f)(\gamma_1)$, we obtain $\forall \gamma_1, \gamma_2 \in \beta_1 : \gamma_1 \neq \gamma_2 \Rightarrow (g \circ f)(\gamma_1) \neq (g \circ f)(\gamma_2) \vee (g \circ f)(\gamma_1) \in \text{Indef}(\beta_3)$.
3. By hypothesis, we have $\text{Def}(\beta_3) \subseteq \text{range}(g)$. By property 1 of Definition 9, there exist $\gamma_1, \dots, \gamma_n$ in $\text{Def}(\beta_2)$ such that $\{f(\gamma_1), \dots, f(\gamma_n)\} = \text{Def}(\beta_3)$. The result follows from the hypothesis that $\text{Def}(\beta_2) \subseteq \text{range}(f)$.

□

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 22.

□

Lemma 23 Let γ be an abstract constraint with multiplicity and θ a concrete constraint store. $\theta \in Cc(\gamma) \Rightarrow \forall \lambda \in \theta : \lambda \in Cc(cons(\gamma))$.

Proof: Immediate consequence of Definition 6. □

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 .

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

Basis: $\beta = \emptyset$. If $\theta \in Cc_i(\beta)$ then $\theta = \emptyset$ by Definition 7 and we choose the empty function as a lifting function. Conversely, if there exists a function $f : \theta \rightarrow \beta$, 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,

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

(\Rightarrow) Let $\theta \in Cc_i(\beta)$. Hence, $\theta = \theta_1 \sqcup \theta'$, 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 $f : \theta \rightarrow \beta$ 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 23
 $\Rightarrow \forall \lambda \in \theta : \lambda \in Cc(cons(f(\lambda)))$.
2. $\forall \lambda_1, \lambda_2 \in \theta' : \lambda_1 \neq \lambda_2 \Rightarrow f'(\lambda_1) \neq f'(\lambda_2) \vee f'(\lambda_1) \in \text{Indef}(\beta')$ by Definition 11
 $\Rightarrow \forall \lambda_1, \lambda_2 \in \theta' : \lambda_1 \neq \lambda_2 \Rightarrow f(\lambda_1) \neq f(\lambda_2) \vee f(\lambda_1) \in \text{Indef}(\beta)$
since $\text{Indef}(\beta') \subseteq \text{Indef}(\beta)$

$\forall \lambda_1 \in \theta_1, \lambda_2 \in \theta', \lambda_1 \neq \lambda_2 \Rightarrow f(\lambda_1) \neq f(\lambda_2)$ 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

$\Rightarrow \forall \lambda_1, \lambda_2 \in \theta_1 : \lambda_1 \neq \lambda_2 \Rightarrow f(\lambda_1) \in \text{Indef}(\beta)$ since $\gamma = f(\lambda_1)$

3. $\text{Def}(\beta') \subseteq \text{range}(f')$ by Definition 11

γ is not definite $\Rightarrow \text{Def}(\beta) = \text{Def}(\beta') \wedge \text{range}(f') \subseteq \text{range}(f)$

$\Rightarrow \text{Def}(\beta) \subseteq \text{range}(f)$

γ is definite $\Rightarrow \text{Def}(\beta) = \{\gamma\} \cup \text{Def}(\beta') \wedge \{\gamma\} \cup \text{range}(f') = \text{range}(f)$

$\Rightarrow \text{Def}(\beta) \subseteq \text{range}(f)$.

(\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 $\lambda_1, \dots, \lambda_m$ be all elements in θ such that $f(\lambda_i) = \gamma$ ($1 \leq i \leq m$). $\lambda_i \in Cc(\text{cons}(\gamma))$ ($1 \leq i \leq m$) by Definition 11 and $\{\lambda_1, \dots, \lambda_m\} \in Cc(\gamma)$ by Definition 6. Let $\theta = \{\lambda_1, \dots, \lambda_m\} \sqcup \theta'$ and consider the function $f' : \theta' \rightarrow \beta'$ defined by $f'(\lambda) = f(\lambda)$ for all $\lambda \in \theta'$. We show that f' is a lifting function of θ' to β' .

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

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

3. $\text{Def}(\beta) \subseteq \text{range}(f)$ by Definition 11

$\text{range}(f') = \text{range}(f) \setminus \{\gamma\}$

$\text{Def}(\beta') = \text{Def}(\beta) \Rightarrow \text{Def}(\beta') \subseteq \text{range}(f)$ since γ is not definite

$\Rightarrow \text{Def}(\beta') \subseteq \text{range}(f')$ since $\gamma \notin \text{Def}(\beta')$.

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

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

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

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

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

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

Lemma 24 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. \square

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. $\forall \lambda \in \theta : \lambda \in Cc(\text{cons}(f(\lambda)))$ by Definition 11
 $\forall f(\lambda) \in \beta_1 : f(\lambda) \sqsubseteq g(f(\lambda))$ by Definition 9
 $\forall f(\lambda) \in \beta_1 : \text{cons}(f(\lambda)) \sqsubseteq \text{cons}(g(f(\lambda)))$ by ordering on AM_D
 $\forall \lambda \in \theta : \lambda \in Cc(\text{cons}(g(f(\lambda))))$ by Lemma 1.
2. $\forall \lambda_1, \lambda_2 \in \theta : \lambda_1 \neq \lambda_2$
 $\Rightarrow f(\lambda_1) \neq f(\lambda_2) \vee f(\lambda_1) \in \text{Indef}(\beta_1)$ by Definition 11
 $\Rightarrow f(\lambda_1) \neq f(\lambda_2) \vee g(f(\lambda_1)) \in \text{Indef}(\beta_2)$ by $f(\lambda_1) \sqsubseteq g(f(\lambda_1))$, Lemma 24
 $\Rightarrow g(f(\lambda_1)) \neq g(f(\lambda_2)) \vee g(f(\lambda_1)) \in \text{Indef}(\beta_2) \vee g(f(\lambda_1)) \in \text{Indef}(\beta_2)$ Def. 9
 $\Rightarrow g(f(\lambda_1)) \neq g(f(\lambda_2)) \vee g(f(\lambda_1)) \in \text{Indef}(\beta_2)$

$$\begin{array}{ll}
3. \quad \text{Def}(\beta_2) \subseteq \text{range}(g) & \text{by Definition 9} \\
\Rightarrow \forall \gamma_2 \in \text{Def}(\beta_2) \exists \gamma_1 \in \beta_1 : g(\gamma_1) = \gamma_2 & \\
\Rightarrow \forall \gamma_2 \in \text{Def}(\beta_2) \exists \gamma_1 \in \text{Def}(\beta_1) : g(\gamma_1) = \gamma_2 & \gamma_1 \sqsubseteq g(\gamma_1) = \gamma_2, \text{ Lemma 24} \\
\forall \gamma_1 \in \text{Def}(\beta_1) \exists \lambda \in \theta : f(\lambda) = \gamma_1 & \text{by Definition 11} \\
\Rightarrow \forall \gamma_2 \in \text{Def}(\beta_2) \exists \lambda \in \theta : g(f(\lambda)) = \gamma_2 & \text{combining the above} \\
\Rightarrow \text{Def}(\beta_2) \subseteq \text{range}(g \circ f) &
\end{array}$$

$\theta \in Cc_i(\beta_2)$ 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 , $\theta \in Cc(\beta_2)$. \square

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 $|\beta_1| + |\text{Def}(\beta_2)|$.

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,

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

We first prove that M is a matching. By Definition 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. $|\beta_1| + |\text{Def}(\beta_2)|$.

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

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

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 $(\gamma_1, \gamma_2^n) \in M_4$ and $f(\gamma_1) = \text{undefined}$ 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 $\gamma_2 \in \text{Def}(\beta_2)$ such that $\gamma_2 \notin \text{range}(f)$. 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.

□

A.2 Addition

Theorem 5 Let β be an abstract store and γ be an abstract constraint with multiplicity.

- (i) If $\theta_1 \in Cc_i(\beta)$ and $\theta_2 \in Cc(\gamma)$, then $\theta_1 \sqcup \theta_2 \in Cc_i(\beta \uplus \gamma)$.
- (ii) If $\theta_1 \in Cc(\beta)$ and $\theta_2 \in Cc(\gamma)$, then $\theta_1 \sqcup \theta_2 \in Cc(\beta \uplus \gamma)$.

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

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

We show that f' is a lifting function of $\theta_1 \sqcup \theta_2$ to $\beta \uplus \gamma$.

1. $\forall \lambda_1 \in \theta_1 : \lambda_1 \in Cc(cons(f'(\lambda_1)))$ by Definition 11 and $f'(\lambda_1) = f(\lambda_1)$
 $\forall \lambda_2 \in \theta_2 : \lambda_2 \in Cc(cons(f'(\lambda_2)))$ by Lemma 23, $\gamma \sqsubseteq f'(\lambda_2)$ and Lemma 1
 $\Rightarrow \forall \lambda \in \theta_1 \sqcup \theta_2 : \lambda \in Cc(cons(f'(\lambda)))$.

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

$\gamma \in \beta \Rightarrow f'(\lambda_2) \in \text{Indef}(\beta \uplus \gamma)$ since $\langle \sigma, Any \rangle$ is indefinite

$\forall \lambda_1, \lambda_2 \in \theta_2 : \lambda_1 \neq \lambda_2 \Rightarrow \gamma$ is indefinite by Definition 6.

$\Rightarrow \forall \lambda_1, \lambda_2 \in \theta_2 : \lambda_1 \neq \lambda_2 \Rightarrow f(\lambda_1) \in \text{Indef}(\beta \uplus \gamma)$

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

Therefore $\theta_1 \sqcup \theta_2 \in Cc_i(\beta \uplus \gamma)$.

(ii) Consider now $\theta_1 \in Cc(\beta)$ and $\theta_2 \in Cc(\gamma)$. By definition of Cc , there exists $\theta_i \in Cc_i(\beta)$ s.t. $\theta_1 \leftrightarrow \theta_i$. By part (i), $\theta_i \sqcup \theta_2 \in Cc_i(\beta \uplus \gamma)$. Since $\theta_1 \sqcup \theta_2 \leftrightarrow \theta_i \sqcup \theta_2$, by definition of Cc $\theta_1 \sqcup \theta_2 \in Cc(\beta \uplus \gamma)$. \square

A.3 Upper Bound

Theorem 6 Let β_1 and β_2 be abstract stores and θ be a constraint store.

$$(i) \theta \in Cc_i(\beta_1) \vee \theta \in Cc_i(\beta_2) \Rightarrow \theta \in Cc_i(\text{UNION}(\beta_1, \beta_2)).$$

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

Proof: (i) We only show that $\theta \in Cc_i(\beta_1) \Rightarrow \theta \in Cc_i(\text{UNION}(\beta_1, \beta_2))$. The case $\theta \in Cc_i(\beta_2) \Rightarrow \theta \in Cc_i(\text{UNION}(\beta_1, \beta_2))$ 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 $\gamma = \langle \sigma, \text{One} \rangle$. By definition of the concretization function, we have that $\theta = \theta_1 \sqcup \theta_2$, $\theta_1 \in Cc(\gamma)$, and $\theta_2 \in Cc_i(\beta'_1)$. By hypothesis, we have that $\theta_2 \in Cc_i(\text{UNION}(\beta'_1, \beta_2))$. We also have that $\theta_1 \in Cc(\langle \sigma, \text{ZeroOrOne} \rangle)$ by monotonicity of Cc . Hence, by Theorem 5 (i), we have $\theta_1 \sqcup \theta_2 \in Cc_i(\text{UNION}(\beta'_1, \beta_2)) \sqcup \langle \sigma, \text{ZeroOrOne} \rangle$.
- *case 2ii:* By hypothesis, we have that $\theta \in Cc_i(\text{UNION}(\beta_1, \beta'_2))$. By definition of the concretization function, we have that $\emptyset \in Cc(\langle \sigma, \text{ZeroOrOne} \rangle)$. Hence, by Theorem 5 (i),

$$\theta \in Cc_i(\text{UNION}(\beta_1, \beta'_2)) \sqcup \langle \sigma, \text{ZeroOrOne} \rangle.$$

- *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 $\gamma = \langle s_0 \text{ op } \sum_{i=1}^n s_i x_i, \mu \rangle$. By definition of the concretization function, we have that $\theta = \theta_1 \sqcup \theta_2$, $\theta_1 \in Cc(\gamma)$, and $\theta_2 \in Cc_i(\beta'_1)$. By hypothesis, we have that $\theta_2 \in Cc_i(\text{UNION}(\beta'_1, \beta'_2))$. By monotonicity of Cc , we have that $\theta_1 \in Cc(\langle (s_0 \sqcup s'_0) \text{ op } \sum_{i=1}^n (s_i \sqcup s'_i) x_i, \mu \sqcup \mu' \rangle)$. 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), $\theta_i \in Cc_i(\text{UNION}(\beta_1, \beta_2))$. By definition of Cc , $\theta \in Cc(\text{UNION}(\beta_1, \beta_2))$. \square

A.4 Projection

Lemma 9 [Gauss] Let $v \in \mathbb{N}$, σ be an abstract constraint such that $\sigma[v] = \oplus$ and $op(\sigma)$ is '=' and β be an abstract store. We have

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

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 $\beta = \{\gamma'\} \cup \beta'$ ($\gamma' \notin \beta'$) and $|\beta'| = n$. Let $\theta \in Cc_i(\beta)$. Then, $\theta = \theta_1^i \sqcup \theta_2^i$ with $\theta_1^i \in Cc(\gamma')$ & $\theta_2^i \in Cc_i(\beta')$. We have

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

By hypothesis, we have

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

By Lemma 7, 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). \square

Lemma 10 [Split Top] Let θ be a constraint store, let β be an abstract store, and $v \in \mathbb{N}$. We have

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

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 $\beta = \{\gamma\} \cup \beta_2$ ($\gamma = \langle \sigma, \mu \rangle \notin \beta_2$) and $|\beta_2| = n$. Let $\theta \in Cc_i(\beta)$. Then, $\theta = \theta_1 \sqcup \theta_2$ with $\theta_1 \in Cc(\gamma)$ & $\theta_2 \in Cc_i(\beta_2)$. By the induction hypothesis, we know that $\theta_2 \in Cc_i(\text{Asplit_top}(\beta_2, v))$. 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 $\mu \sqsubseteq \mu \sqcup \text{ZeroOrOne}$. Similarly $\theta_1^+ \in Cc(\gamma^+)$ and $\theta_1^- \in Cc(\gamma^-)$. By Theorem 5 (i), we have

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

The result then follows from Lemma 5 (i). \square

Lemma 11 [Split Basic] Let θ be a constraint store, let β be an abstract store, and $v \in \mathbb{N}$. Let $\sigma[v] \in \{0, \oplus, \ominus\}$ for all constraints $\langle \sigma, \mu \rangle \in \beta$. We have

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

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 $\beta = \{\gamma\} \cup \beta_2$ ($\gamma \notin \beta_2$) and $|\beta_2| = n$. Let $\theta \in Cc_i(\beta)$. Then, $\theta = \theta_1 \sqcup \theta_2$ 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

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

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 $\lambda[v] = \oplus$, $\theta^+ = \theta_2^+ \sqcup \theta_1$ and $\beta^+ = \beta_2^+ \sqcup \gamma$. The result follows by Theorem 5 (i).

Case $\sigma[v] = \ominus$ and $\text{op}(\sigma)$ is not '='. $\theta^+ = \theta_2^+$ and the result follows from the fact that $\beta^+ = \beta_2^+$.

Case $\sigma[v] = \ominus$ and $\text{op}(\sigma)$ is '='. θ_1 contains only constraints such that $\lambda[v] = \ominus$, $\theta^+ = \theta_2^+ \sqcup \theta_3$ where $\theta_3 = \{\text{Cneg}(\lambda) \mid \lambda \in \theta_1\}$ and $\beta^+ = \beta_2^+ \sqcup \gamma_3$ where $\gamma_3 = \langle \text{Aneg}(\sigma), \mu \rangle$. Also, by Lemma 6, $\theta_1 \in Cc(\langle \sigma, \mu \rangle)$ 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). \square

Lemma 12 [Split] Let θ be a constraint store, let β be an abstract store, and $v \in \mathbb{N}$. We have

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

Proof: This follows from Lemma 10, the fact that for all $\gamma = \langle \sigma, \mu \rangle \in \text{Asplit_top}(\beta, v)$, $\sigma[v] \in \{0, \oplus, \ominus\}$ and the consistency of Asplit_basic . \square

Lemma 13 [Fourier Step] Let $v \in \mathbb{N}$, let β^+ be an abstract store such that for all $\gamma \in \beta^+$, $\gamma = \langle \sigma, \mu \rangle$ and $\sigma[v] = \oplus$, let β^- be an abstract store such that for all $\gamma \in \beta^-$, $\gamma = \langle \sigma, \mu \rangle$ and $\sigma[v] = \ominus$. We have

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

Proof: Let β^+ be $\{\gamma_1^+, \dots, \gamma_n^+\}$ and β^- be $\{\gamma_1^-, \dots, \gamma_m^-\}$. Let $\{(\gamma_1^+, \gamma_1^-), \dots, (\gamma_n^+, \gamma_m^-)\}$ 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 8, we have that any pair of constraints $\lambda^+ \in \theta^+$ and $\lambda^- \in \theta^-$ satisfies

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

Moreover, the number of constraints in $\text{Cfourier_step}(\theta^+, \theta^-)$ 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,

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

The result follows from Theorem 5 (i). □

Lemma 14 [Fourier] Let $v \in \mathbb{N}$, θ be a store, and β be an abstract store. We have

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

Proof: This follows from the consistency of *Asplit* (Lemma 12) and of *Afourier_step* (Lemma 13) and from the definition of the concretization which specifies that, when γ is of the form $\langle \sigma, \mu \rangle$ with $\mu \in \{\text{One}, \text{ZeroOrOne}\}$, any $\theta \in Cc(\gamma)$ is of cardinality at most one. □

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

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

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 $\mu = \text{One}$, $\text{op}(\sigma)$ is '=' and $\sigma[v] = \oplus$, then there exists a λ s.t. $\theta_i = \{\lambda\} \sqcup \theta'$, $\lambda \in Cc(\gamma)$,

and $\theta' \in Cc_i(\beta')$. By definition of the concretization, $\lambda[v] > 0$ and it follows from Lemma 9 that

$$C_{\text{gauss}}(\theta', \lambda, v) \in Cc_i(A_{\text{gauss}}(\beta', \sigma, v))$$

and thus

$$C_{\text{project}}(\theta_i, v) \in Cc_i(A_{\text{project}}(\beta, v))$$

Since $C_{\text{project}}(\theta, v) \leftrightarrow C_{\text{project}}(\theta_i, v)$, it follows by definition of Cc that

$$C_{\text{project}}(\theta, v) \in Cc(A_{\text{project}}(\beta, v))$$

The case $\mu = \text{One}$, $\text{op}(\sigma)$ is '=' and $\sigma[v] = \ominus$ is similar. In the last case, we have by Lemma 14 that

$$C_{\text{fourier}}(\theta_i, v) \in Cc_i(A_{\text{fourier}}(\beta, v))$$

Moreover, since $C_{\text{project}}(\theta_i, v) \leftrightarrow C_{\text{fourier}}(\theta_i, v)$, and $C_{\text{project}}(\theta, v) \leftrightarrow C_{\text{project}}(\theta_i, v)$, it follows by definition of Cc that

$$C_{\text{project}}(\theta, v) \in Cc(A_{\text{project}}(\beta, v))$$

□

A.5 Satisfiability

Lemma 25 [Is Trivially Satisfiable] Let θ be a store and β be an abstract store.

- (i) $\theta \in Cc_i(\beta) \Rightarrow (A_{\text{is_triv_sat}}(\beta) \Rightarrow C_{\text{is_triv_sat}}(\theta))$
- (ii) $\theta \in Cc(\beta) \wedge \text{Var}(\theta) = \emptyset \wedge \text{Var}(\beta) = \emptyset$
 $\Rightarrow (A_{\text{is_triv_sat}}(\beta) \Rightarrow C_{\text{is_triv_sat}}(\theta))$

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 $\beta = \{\gamma\} \cup \beta'$ ($\gamma \notin \beta'$) with $\gamma = \langle \sigma, \mu \rangle$ and $|\beta'| = n$. Let $\theta \in Cc_i(\beta)$. Then, $\theta = \theta_1 \sqcup \theta_2$ 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 $C_{\text{is_triv_sat}}(\theta_1)$ is true. This gives

$$C_{\text{is_triv_sat}}(\theta) = C_{\text{is_triv_sat}}(\theta_1) \wedge C_{\text{is_triv_sat}}(\theta_2) = C_{\text{is_triv_sat}}(\theta_2)$$

But $\text{A_is_triv_sat}(\beta) = \text{A_is_triv_sat}(\beta')$. 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 $\text{A_is_triv_sat}(\beta)$ 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 $\text{Var}(\beta) = \emptyset$ gives $\text{Var}(\theta_i) = \emptyset$. Moreover $\text{Var}(\theta) = \emptyset$. This implies that $\text{C_is_triv_sat}(\theta) = \text{C_is_triv_sat}(\theta_i)$. The desired result then follows from part (i). \square

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

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

Proof: Let $\theta \in Cc(\beta)$. By Corollary 1, we have that $\theta_p \in Cc(\beta_p)$. Also, $\text{Var}(\beta_p) = \emptyset$ and $\text{Var}(\theta_p) = \emptyset$. The desired result follows by Lemma 25 (ii). \square

A.6 Unsatisfiability

Lemma 26 [Is Trivially Unsatisfiable] Let θ be a store and β be an abstract store.

$$(i) \theta \in Cc_i(\beta) \Rightarrow (\text{Ais_triv_unsat}(\beta) \Rightarrow \text{Cis_triv_unsat}(\theta))$$

$$(ii) \theta \in Cc(\beta) \wedge \text{Var}(\theta) = \emptyset \wedge \text{Var}(\beta) = \emptyset$$

$$\Rightarrow (\text{Ais_triv_unsat}(\beta) \Rightarrow \text{Cis_triv_unsat}(\theta))$$

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 $\beta = \{\gamma\} \cup \beta'$ ($\gamma \notin \beta'$) with $\gamma = \langle \sigma, \mu \rangle$ and $|\beta'| = n$. Let $\theta \in Cc_i(\beta)$. Then, $\theta = \theta_1 \sqcup \theta_2$ with $\theta_1 \in Cc(\gamma)$ & $\theta_2 \in Cc_i(\beta')$.

Case σ is $s_0 < \sum_{i=1}^n 0x_i \wedge (s_0 = 0 \vee s_0 = \oplus) \wedge \mu = \text{One}$. Here $\text{Ais_triv_unsat}(\beta)$ 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 $\text{C_is_triv_unsat}(\theta_1)$ 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. $\text{A_is_triv_unsat}(\beta) = \text{A_is_triv_unsat}(\beta')$. If $\text{Cis_triv_unsat}(\theta)$ is true, there is nothing to prove. Otherwise $\text{Cis_triv_unsat}(\theta) = \text{Cis_triv_unsat}(\theta')$, 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 $\text{Var}(\beta) = \emptyset$ gives $\text{Var}(\theta_i) = \emptyset$. Moreover $\text{Var}(\theta) = \emptyset$. This implies that $\text{C_is_triv_unsat}(\theta) = \text{C_is_triv_unsat}(\theta_i)$. The desired result then follows from part (i). \square

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

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

Proof: Let $\theta \in Cc(\beta)$. By Corollary 1, we have that $\theta_p \in Cc(\beta_p)$. Also, $\text{Var}(\beta_p) = \emptyset$ and $\text{Var}(\theta_p) = \emptyset$. The desired result follows by Lemma 26 (ii). \square

A.7 Conditional Satisfiability

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

- (i) $\theta \in Cc_i(\beta) \Rightarrow \text{Cred_gnd}(\theta) \in Cc_i(\text{Ared_gnd}(\beta))$.
- (ii) $\theta \in Cc(\beta) \wedge \theta \text{ satisfiable} \Rightarrow \text{Cred_gnd}(\theta) \in Cc(\text{Ared_gnd}(\beta))$.
- (iii) $\theta \in Cc(\beta) \wedge \theta \text{ satisfiable} \Rightarrow \theta \in Cc(\text{Ared_gnd}(\beta))$.

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 $\{\gamma\} \cup \beta'$ ($\gamma \notin \beta'$) with $\gamma = \langle \sigma, \mu \rangle$ and $|\beta'| = n$. Let $\theta \in Cc_i(\beta)$. Then $\theta = \theta_1 \sqcup \theta_2$ with $\theta_1 \in Cc(\gamma)$ and $\theta_2 \in Cc_i(\beta')$. According to the induction hypothesis, we have

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

Case σ is $s_0 \text{ op } \sum_{i=1}^n 0x_i$. By definition of the concretization function for abstract constraints with multiplicity, we have that θ_1^i consists only of constraints of the form $c_0 \text{ op } \sum_{i=1}^n 0x_i$ and therefore $\text{Cred_gnd}(\theta_1) = \emptyset$. This gives

$$\text{Cred_gnd}(\theta) = \text{Cred_gnd}(\theta_1) \sqcup \text{Cred_gnd}(\theta_2) = \text{Cred_gnd}(\theta_2).$$

But $\text{Ared_gnd}(\beta) = \text{Ared_gnd}(\beta')$. The desired result follows from the induction hypothesis.

Case σ is not $s_o \text{ op } \sum_{i=1}^n 0x_i$. Again by definition of the concretization function for abstract constraints with multiplicity, we have that $\text{Cred_gnd}(\theta_1) = \theta_1$. This gives $\text{Cred_gnd}(\theta) = \text{Cred_gnd}(\theta_2) \sqcup \theta_1$. But $\text{Ared_gnd}(\beta) = \text{Ared_gnd}(\beta') \uplus \gamma$. 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

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

But θ satisfiable implies that $\text{Cred_gnd}(\theta) \leftrightarrow \text{Cred_gnd}(\theta_i)$. 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 $\theta \leftrightarrow \text{Cred_gnd}(\theta)$. \square

Lemma 28 [Reduce Equations Step] Let $v \in \aleph$, θ be a store and β be an abstract store.

$$(i) \theta \in Cc(\beta) \Rightarrow \text{Cred_eqn_step}(\theta, v) \in Cc(\text{Ared_eqn_step}(\beta, v)).$$

$$(ii) \theta \in Cc(\beta) \Rightarrow \theta \in Cc(\text{Ared_eqn_step}(\beta, v)).$$

Proof: (i) If $\beta = \emptyset$ the result is obvious. Otherwise, let $\beta = \{\gamma\} \cup \beta'$ ($\gamma \notin \beta'$) 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 $\mu = \text{One} \wedge \text{op}(\sigma)$ is '=' $\wedge \sigma[v] = \oplus$. There exists a λ s.t. $\theta_i = \{\lambda\} \sqcup \theta'$, $\lambda \in Cc(\sigma)$, and $\theta' \in Cc_i(\beta')$. By definition of the concretization, $\lambda[v] > 0$ and it follows from Lemma 9 that

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

and thus by definition of Cc that

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

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

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

Case $\mu = \text{One} \wedge \text{op}(\sigma)$ is '=' $\wedge \sigma[v] = \ominus$. The arguments are very similar.

Otherwise. We have that $\text{Ared_eqn_step}(\beta, v) = \beta$. Moreover, $\theta \leftrightarrow \text{Cred_eqn_step}(\theta, v)$. 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 $\theta \leftrightarrow \text{Cred_eqn_step}(\theta, v)$. \square

Lemma 29 [Reduce Equations Set] Let $V \in 2^{\mathbb{N}}$, θ be a store and β be an abstract store.

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

Proof: The proof is a simple induction of applying Lemma 28 (ii) $|V|$ times. \square

Lemma 30 [Reduce Equations] Let θ be a store and β be an abstract store.

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

Proof: Direct consequence of Lemma 29. \square

Lemma 31 [Reduce Tops] Let θ be a store and β be an abstract store.

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

Proof: If $\beta = \emptyset$ or $|\beta| = 1$, the result is obvious. Otherwise, let $\beta = \{\gamma_1, \gamma_2\} \cup \beta'$ ($\gamma_1 \notin \beta'$, $\gamma_2 \notin \beta'$) with $\gamma_1 = \langle \sigma_1, \mu_1 \rangle$ and $\gamma_2 = \langle \sigma_2, \mu_2 \rangle$. 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 $\mu = \text{One} \wedge (\sigma_1 \text{ is } \oplus \leq \oplus x_v) \wedge (\sigma_2 \text{ is } \top = \oplus x_v)$. There exists a λ_1 such that $\theta_i = \{\lambda_1\} \sqcup \theta_2 \sqcup \theta'$, $\lambda_1 \in Cc(\sigma_1)$, $\theta_2 \in Cc(\gamma_2)$ 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 $\{\lambda_1\} \sqcup \theta_2$ 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 concretization 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 $\theta_i \in Cc_i(\text{Ared_top}(\beta))$. The desired result follows by the definition of Cc .

Other subcases and second case. Proof is similar.

Otherwise. We have that $\text{Ared_top}(\beta) = \beta$ and so there is nothing to prove. \square

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

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

Proof: Consequence of Lemma 31, Lemma 30, and Lemma 27 (iii) □

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

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

Proof:

$$\begin{aligned} & \text{Ais_cond_sat}(\beta_1, \beta_2) \\ &= \text{Ais_unsat}(\beta_1) \vee \text{Ais_sat}(\text{Areduce}(\text{Aproject_set}(\beta_1, \text{Var}(\beta_1) \setminus \text{Var}(\beta_2))) \sqcup \beta_2). \end{aligned}$$

And its concrete counterpart

$$\text{Cis_cond_sat}(\theta_1, \theta_2) = \text{Cis_unsat}(\theta_1) \vee \text{Cis_sat}(\theta_1 \sqcup \theta_2).$$

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

$$\begin{aligned} & \text{Cis_sat}(\theta_1 \sqcup \theta_2) \\ &= \text{Cis_triv_sat}(\text{Cproject_set}(\theta_1 \sqcup \theta_2, \aleph)) \\ &= \text{Cis_triv_sat}(\text{Cproject_set}(\text{Cproject_set}(\theta_1, V) \sqcup \theta_2, \aleph)) \\ &= \text{Cis_sat}(\text{Cproject_set}(\theta_1, V) \sqcup \theta_2) \end{aligned}$$

By Corollary 1 and Theorem 10, we have that

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

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

A.8 Redundancy

In this section, we shall use the nondeterminism in the concrete redundancy algorithm to prove that there is a concrete algorithm which the abstract algorithm mimics. Since any of the concrete algorithms is correct (it does not matter what choices are made by the concrete algorithm), the correctness of the abstract algorithm follows. In the proof of consistency of `Asimplify_step`, it is necessary to assume that the eliminating constraint chosen by the concrete algorithm is the one corresponding to the abstract constraint chosen by the abstract algorithm.

Lemma 32 [Simplify Step] Let $v \in \mathbb{N}$, θ_r, θ be stores and β_r, β be abstract stores. Then

$$\theta_r \in Cc(\beta_r) \wedge \theta \in Cc(\beta) \Rightarrow \text{Csimplify_step}(\theta_r, \theta, v) \in Cc(\text{Asimplify_step}(\beta_r, \beta, v))$$

Proof: Let $\theta \in Cc(\beta) \wedge \theta_r \in Cc(\beta_r)$. If $\beta = \emptyset$ the result is obvious. Otherwise, let $\beta = \{\gamma\} \cup \beta'$ ($\gamma \notin \beta'$) with $\gamma = \langle \sigma, \mu \rangle$. Then, by definition of Cc , there exists $\theta_i \in Cc_i(\beta)$ s.t. $\theta \leftrightarrow \theta_i$ and there exists $\theta'_i \in Cc_i(\beta_r)$ s.t. $\theta_r \leftrightarrow \theta'_i$

Case $\mu = \text{One} \wedge \text{op}(\sigma)$ is $'=' \wedge \sigma[v] = \oplus$. There exists a λ s.t. $\theta_i = \{\lambda\} \sqcup \theta'$, $\lambda \in Cc(\sigma)$, and $\theta' \in Cc_i(\beta')$. By definition of the concretization, $\lambda[v] > 0$ and it follows from Lemma 9 that

$$\text{Cgauss}(\theta'_i, \lambda, v) \in Cc_i(\text{Agauss}(\beta_r, \sigma, v))$$

and thus by definition of Cc that

$$\text{Cgauss}(\theta_r, \lambda, v) \in Cc(\text{Agauss}(\beta_r, \sigma, v))$$

The desired result follows immediately.

Case $\mu = \text{One} \wedge \text{op}(\sigma)$ is $'=' \wedge \sigma[v] = \ominus$. The arguments are very similar.

Otherwise. We have that $\text{Asimplify_step}(\beta_r, \beta, v) = \beta_r$. There is a concrete algorithm such that $\text{Csimplify_step}(\theta_r, \theta, v) = \theta_r$. The desired result follows immediately. \square

Lemma 33 [Simplify] Let $V \in 2^{\mathbb{N}}$, θ_r, θ be stores and β_r, β be abstract stores.

$$\theta_r \in Cc(\beta_r) \wedge \theta \in Cc(\beta) \Rightarrow \text{Csimplify}(\theta_r, \theta, V) \in Cc(\text{Asimplify}(\beta_r, \beta, V))$$

Proof: The proof is a simple application of Lemma 32. \square

To prove the consistency of `Ais_redundant`, it is necessary to work with a modified concrete algorithm that projects exactly the same variables as the abstract algorithm before it calls `Creduce`. This is because $\text{Var}(\theta_r) \subseteq \text{Var}(\beta_r)$ and so the concrete algorithm as presented in the main text may project more variables than the abstract algorithm.

Theorem 12 [Is Redundant] Let θ_r, θ be stores and β_r, β be abstract stores.

$$\begin{aligned} & \theta_r \in Cc(\beta_r) \wedge \theta \in Cc(\beta) \wedge \theta \text{ satisfiable} \\ & \Rightarrow (\text{Ais_redundant}(\beta_r, \beta) \Rightarrow \text{Cis_redundant}(\theta_r, \theta)) \end{aligned}$$

Proof: There is a concrete redundancy algorithm that projects exactly the same variables as the abstract algorithm before calling `Creduce` and therefore by Corollary 1 and

Theorem 10, $\theta_p \in Cc(\beta_p)$. By the consistency of `Asimplify` (Lemma 33), we have that

$$\theta_s \in Cc(\beta_s)$$

If $\text{Var}(\beta_s) \neq \emptyset$ then there is nothing to prove as `Ais_triv_sat`(β_s) is false. Otherwise $\text{Var}(\beta_s) = \emptyset$ and $\theta_s \in Cc(\beta_s)$ gives $\text{Var}(\theta_s) = \emptyset$. The result then follows from the consistency of `Ais_triv_sat` (Lemma 25(ii)). \square

A.9 Freeness

Theorem 13 [Is Free] Let $v \in \mathbb{N}$, θ be a constraint store and β be an abstract store.

- (i) $\theta \in Cc_i(\beta) \Rightarrow (\text{Ais_free}(\beta, v) \Rightarrow \text{Cis_free}(\theta, v))$.
- (ii) $\theta \in Cc(\beta) \wedge \theta \text{ satisfiable} \Rightarrow (\text{Ais_free}(\beta, v) \Rightarrow \text{Cis_free}(\theta, v))$.

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 $\beta = \{\gamma\} \cup \beta'$ ($\gamma \notin \beta'$) with $\gamma = \langle \sigma, \mu \rangle$ and $|\beta'| = n$. Let $\theta \in Cc_i(\beta)$. Then, $\theta = \theta_1 \sqcup \theta_2$ with $\theta_1 \in Cc(\gamma)$ & $\theta_2 \in Cc_i(\beta')$.

Case $\sigma[v] = 0$. By definition of the concretization for abstract constraints, $\lambda[v] = 0$ for each $\lambda \in \theta_1$. Therefore $\text{Cis_free}(\theta, v) = \text{Cis_free}(\theta_1, v) \wedge \text{Cis_free}(\theta_2, v) = \text{Cis_free}(\theta_2, v)$, as $\text{Cis_free}(\theta_1, v)$ is true. But $\text{Ais_free}(\beta, v) = \text{Ais_free}(\beta', v)$. The desired result then follows by the induction hypothesis.

Otherwise. There is nothing to prove because $\text{Ais_free}(\beta, v)$ is false.

- (ii) Let $\theta \in Cc(\beta)$. By definition of Cc , there exists $\theta_i \in Cc_i(\beta)$ s.t. $\theta \leftrightarrow \theta_i$. By part (i),

$$\text{Ais_free}(\beta, v) \Rightarrow \text{Cis_free}(\theta_i, v)$$

But θ satisfiable and $\theta \leftrightarrow \theta_i$ gives

$$\text{Cis_free}(\theta, v) = \text{Cis_free}(\theta_i, v)$$

The desired result follows immediately. \square

A.10 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, $\text{n_infty}(\alpha)$ denotes the number of infinity symbols in the abstract multistore α . We also introduce the function 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. $\text{n_infty_zero}(\alpha) = \langle \text{n_infty}(\alpha), \text{n_zero}(\alpha) \rangle$. For each n , we consider the set of normalized abstract multistores χ_n over the variables $\{x_1, \dots, x_n\}$. 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 $\{x_1, \dots, x_n\}$ (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

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

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

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

The set

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

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

$$\begin{aligned} & \text{measure}(\alpha_1) < \text{measure}(\alpha_2) \\ \Leftrightarrow & (|\alpha_1| < |\alpha_2|) \vee ((|\alpha_1| = |\alpha_2|) \wedge (\text{n_infty_zero}(\alpha_1) < \text{n_infty_zero}(\alpha_2))) \end{aligned}$$

where

$$\begin{aligned} & \text{n_infty_zero}(\alpha_1) < \text{n_infty_zero}(\alpha_2) \\ \Leftrightarrow & (\text{n_infty}(\alpha_1) < \text{n_infty}(\alpha_2)) \vee \\ & ((\text{n_infty}(\alpha_1) = \text{n_infty}(\alpha_2)) \wedge (\text{n_zero}(\alpha_1) < \text{n_zero}(\alpha_2))) \end{aligned}$$

The following lemmas are enhanced versions of the lemmas in the main text.

Lemma 15 Let s_{old} and s_{new} be intervals s.t. $\text{shape}(s_{new}) = \text{shape}(s_{old})$. Then

- (i) $\text{shape}(s_{old}) = \text{shape}(s_{new} \nabla s_{old})$.
- (ii) $c \in Cc(s_{old}) \vee c \in Cc(s_{new}) \Rightarrow c \in Cc(s_{new} \nabla s_{old})$.
- (iii) $\text{n_infty_zero}(s_{old}) \leq \text{n_infty_zero}(s_{new} \nabla s_{old})$.
- (iv) $s_{new} \not\sqsubseteq s_{old} \Rightarrow \text{n_infty_zero}(s_{old}) < \text{n_infty_zero}(s_{new} \nabla s_{old})$.

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

If $\text{neg}(s_{old})$. The arguments are similar about the right endpoint.

If $\text{zer}(s_{old})$. $\text{zer}(s_{new})$ and $s_{new} \nabla s_{old} = s_{old}$.

Otherwise. $\text{shape}(s_{old}) = \top$. We have that $\text{left}(s_{new} \nabla s_{old}) = \text{left}(s_{old}) \leq 0$ and $\text{open_left}(s_{new} \nabla s_{old}) = \text{open_left}(s_{old})$ or $\text{left}(s_{new} \nabla s_{old}) = -\infty$ and

$\text{open_left}(s_{new} \nabla s_{old}) = \text{true}$. This implies $\neg \text{pos}(s_{new} \nabla s_{old})$. It can be similarly proved that $\neg \text{neg}(s_{new} \nabla s_{old})$, by considering the right endpoint. It can also be proved that $\neg \text{zer}(s_{new} \nabla s_{old})$ by considering both endpoints. This gives $\text{shape}(s_{new} \nabla s_{old}) = \top$.

(ii) If $\neg \text{decrease}(s_{new}, s_{old})$. We have that the left limit of $s_{new} \nabla s_{old}$ is the same as that of s_{old} and therefore $\neg \text{decrease}(s_{old}, s_{new} \nabla s_{old})$ and $\neg \text{decrease}(s_{new}, s_{new} \nabla s_{old})$.

If $\text{decrease}(s_{new}, s_{old}) \wedge \text{pos}(s_{old})$. The left limit of $s_{new} \nabla s_{old}$ is the minimum possible for a positive interval and so $\neg \text{decrease}(s_{old}, s_{new} \nabla s_{old})$ and $\neg \text{decrease}(s_{new}, s_{new} \nabla s_{old})$.

If $\text{decrease}(s_{new}, s_{old}) \wedge \neg \text{pos}(s_{old})$. The left limit of $s_{new} \nabla s_{old}$ is the minimum possible for any interval and so $\neg \text{decrease}(s_{old}, s_{new} \nabla s_{old})$ and $\neg \text{decrease}(s_{new}, s_{new} \nabla s_{old})$.

In all cases, we have that $\neg \text{decrease}(s_{old}, s_{new} \nabla s_{old})$ and $\neg \text{decrease}(s_{new}, s_{new} \nabla s_{old})$. Considering the right endpoint, we can similarly prove that $\neg \text{increase}(s_{old}, s_{new} \nabla s_{old})$ and $\neg \text{increase}(s_{new}, s_{new} \nabla s_{old})$.

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 $\neg \text{decrease}(s_{new}, s_{old})$) or become an infinity symbol (if $\text{decrease}(s_{new}, s_{old}) \wedge \neg \text{pos}(s_{old})$) or become a zero (if $\text{decrease}(s_{new}, s_{old}) \wedge \text{pos}(s_{old})$). Also in the last case, the left endpoint cannot be

originally an infinity symbol because $\text{pos}(s_{old})$. Similar arguments apply at the right endpoint of an interval. The desired result follows.

(iv) By definition of the ordering on intervals,

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

i.e.

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

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

Lemma 16 Let γ_{old} and γ_{new} be abstract constraints with multiplicity s.t. $\text{shape}(\gamma_{new}) = \text{shape}(\gamma_{old})$. Then

- (i) $\text{shape}(\gamma_{old}) = \text{shape}(\gamma_{new} \nabla \gamma_{old})$.
- (ii) $\theta \in Cc(\gamma_{old}) \vee \theta \in Cc(\gamma_{new}) \Rightarrow \theta \in Cc(\gamma_{new} \nabla \gamma_{old})$.
- (iii) $\text{n_infy_zero}(\gamma_{old}) \leq \text{n_infy_zero}(\gamma_{new} \nabla \gamma_{old})$.
- (iv) $\gamma_{new} \not\sqsubseteq \gamma_{old} \Rightarrow \text{n_infy_zero}(\gamma_{old}) < \text{n_infy_zero}(\gamma_{new} \nabla \gamma_{old})$.

Proof: (i) This is a direct consequence of Lemma 15 (i).

(ii) This is a direct consequence of Lemma 15 (ii) and the definition of the concretization of an abstract constraint with multiplicity.

(iii) Direct consequence of Lemma 15 (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 15 (iv) in conjunction with part (iii) leads to the desired result. \square

Lemma 17 Let β_{old} and β_{new} be normalized abstract stores s.t. $\text{shape}(\beta_{new}) = \text{shape}(\beta_{old})$. Then

- (i) $\text{shape}(\beta_{old}) = \text{shape}(\beta_{new} \nabla \beta_{old})$ and $\beta_{new} \nabla \beta_{old}$ is normalized.
- (ii) $\theta \in Cc(\beta_{old}) \vee \theta \in Cc(\beta_{new}) \Rightarrow \theta \in Cc(\beta_{new} \nabla \beta_{old})$.

$$(iii) \text{ n_infy_zero}(\beta_{old}) \leq \text{ n_infy_zero}(\beta_{new} \nabla \beta_{old}).$$

$$(iv) \beta_{new} \not\sqsubseteq \beta_{old} \Rightarrow \text{ n_infy_zero}(\beta_{old}) < \text{ n_infy_zero}(\beta_{new} \nabla \beta_{old}).$$

Proof: (i) The result is a direct consequence of Lemma 16 (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

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

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

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

By the induction hypothesis and Lemma 16 (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 16 (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

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

Applying Lemma 16 (iv) in conjunction with part (iii) leads to the desired result. \square

Lemma 18 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) $|\alpha_{old}| \leq |\alpha_{new} \nabla \alpha_{old}|$.

$$\begin{aligned}
& \text{(iv)} \ (\alpha_{new} \not\sqsubseteq \alpha_{old}) \wedge (|\alpha_{old}| = |\alpha_{new} \nabla \alpha_{old}|) \\
& \Rightarrow \text{n_infty_zero}(\alpha_{old}) < \text{n_infty_zero}(\alpha_{new} \nabla \alpha_{old})
\end{aligned}$$

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 $\text{shape}(\beta_{new}) \not\sqsubseteq \text{shape}(\alpha_{old})$, the result follows immediately. Otherwise, there exists a $\beta_{old} \in \alpha_{old}$ such that $\text{shape}(\beta_{new}) = \text{shape}(\beta_{old})$ and by Lemma 17 (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

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

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 17 (iii) that

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

Also

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

The desired result then follows by applying Lemma 17 (iv) in conjunction with part (iii). \square

Theorem 15 Operation ∇ is a widening operator.

Proof: By Lemma 18 (ii),

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

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

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

We show that $\alpha_0, \dots, \alpha_i, \dots$ is stationary. Denote α_i by α_{old} and $\text{normal}(\alpha'_i)$ by α_{new} . If $\text{normal}(\alpha'_i) \sqsubseteq \alpha_i$, then by definition of ∇ , $\alpha_{i+1} = \alpha_i$. Otherwise, since $\text{normal}(\alpha'_i) \not\sqsubseteq \alpha_i$, by Lemma 18 (iii) and (iv),

$$\begin{aligned}(|\alpha_{old}| < |\alpha_{new} \nabla \alpha_{old}|) \vee \\ ((|\alpha_{old}| = |\alpha_{new} \nabla \alpha_{old}|) \wedge (\text{n_infty_zero}(\alpha_{old}) < \text{n_infty_zero}(\alpha_{new} \nabla \alpha_{old})))\end{aligned}$$

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

A.11 Reordering

Lemma 20 Let P be a program. Then for every satisfiable constraint store θ , every constraint λ and every body G ,

$$\lambda \text{ FF } \tau_P(\langle G \Diamond \theta \rangle) \Rightarrow \tau_P(\langle G \Diamond \theta \rangle) \approx \tau_P(\langle G \Diamond \theta \sqcup \{\lambda\} \rangle).$$

Proof: The proof consists of proving that for any satisfiable constraint store θ , constraint λ , body G and finite depth d s.t. $\lambda \text{ FF } \tau_P(\langle G \Diamond \theta \rangle)$:

$$\tau_P(\langle G \Diamond \theta \rangle) \approx_d \tau_P(\langle G \Diamond \theta \sqcup \{\lambda\} \rangle).$$

The proof proceeds by induction on d .

Basis: $d = 1$. For any body G , constraint λ and satisfiable store θ s.t. $\lambda \text{ FF } \tau_P(\langle G \Diamond \theta \rangle)$, we have that $\theta \sqcup \{\lambda\}$ is also satisfiable. This gives $\langle G \Diamond \theta \rangle \approx \langle G \Diamond \theta \sqcup \{\lambda\} \rangle$, and it follows from Definition 38 that

$$\tau_P(\langle G \Diamond \theta \rangle) \approx_1 \tau_P(\langle G \Diamond \theta \sqcup \{\lambda\} \rangle).$$

Induction Step: Assume that the hypothesis holds for similarity of depth not greater than d , for every G , θ , and λ s.t. $\lambda \text{ FF } \tau_P(\langle G \Diamond \theta \rangle)$. We show that it holds for similarity of depth $d + 1$.

Case $G = \epsilon$.

$$\begin{aligned}\tau_P(\langle G \diamond \theta \rangle) &= \text{tree}(\langle G \diamond \theta \rangle, []) \\ \tau_P(\langle G \diamond \theta \sqcup \{\lambda\} \rangle) &= \text{tree}(\langle G \diamond \theta \sqcup \{\lambda\} \rangle, [])\end{aligned}$$

The similarity to depth $d + 1$ follows easily from Definition 38 because the child list is empty for both trees.

Case $G = \langle \theta', q \rangle :: B$ and $\theta \sqcup \theta'$ is inconsistent or no clause in P has head q .

We must have that $\theta \sqcup \{\lambda\} \sqcup \theta'$ is also inconsistent or no clause in P has head q . The proof is then similar to the previous case as the child list is empty for both trees.

Otherwise.

Let $G = \langle \theta', q \rangle :: B$.

$$\tau_P(\langle G \diamond \theta \rangle) = \text{tree}(\langle G \diamond \theta \rangle, [T_1, \dots, T_n])$$

where $H_i : -B_i$ is the i^{th} of the n clauses of P with head q (renamed with all new variables) and

$$T_i = \tau_P(\langle B_i :: B \diamond \theta \sqcup \theta' \sqcup (q = H_i) \rangle)$$

By Lemma 20 we have that λ is failure free in each T_i and so

$$\forall i : \theta \sqcup \theta' \sqcup (q = H_i) \text{ consistent} \Rightarrow \theta \sqcup \{\lambda\} \sqcup \theta' \sqcup (q = H_i) \text{ consistent}$$

This gives us that

$$\tau_P(\langle G \diamond \theta \sqcup \{\lambda\} \rangle) = \text{tree}(\langle G \diamond \theta \sqcup \{\lambda\} \rangle, [T'_1, \dots, T'_n])$$

where

$$T'_i = \tau_P(\langle B_i :: B \diamond \theta \sqcup \{\lambda\} \sqcup \theta' \sqcup (q = H_i) \rangle)$$

By the induction hypothesis,

$$\forall i : T_i \approx_d T'_i.$$

Moreover, $\langle G \diamond \theta \rangle \approx \langle G \diamond \theta \sqcup \{\lambda\} \rangle$. The desired result follows from Definition 38. \square

Before proving the main theorem, it is convenient to give a precise definition for the inclusive and exclusive execution of queries. This is done by defining two operational semantics for any program containing the syntactic construct $?$. One corresponding to $\theta ? \{\lambda\} = \theta$ is called the exclusive operational semantics, while the other corresponding to $\theta ? \{\lambda\} = \theta \sqcup \{\lambda\}$ is called the inclusive operational semantics.

Definition 47 [Exclusive Operational Semantics $\tau_P^x : \text{State} \mapsto \text{Tree}$] The exclusive operational semantics of a program P is a mapping $\tau_P^x : \text{State} \mapsto \text{Tree}$ defined as follows:

$$\tau_P^x(\langle G \Diamond \theta \rangle) = \text{tree}(\langle G \Diamond \theta \rangle, L)$$

where

$$L = \left\{ \begin{array}{ll} [] & \text{if } G = \epsilon \\ [] & \text{if } G = \langle \theta', q \rangle :: B \text{ and} \\ & \theta \sqcup \theta' \text{ is inconsistent or no clause in } P \text{ has head } q \\ [] & \text{if } G = \langle \theta' ? \{ \lambda \}, q \rangle :: B \text{ and} \\ & \theta \sqcup \theta' \text{ is inconsistent or no clause in } P \text{ has head } q \\ [T_1, \dots, T_n] & \text{if } G = \langle \theta', q \rangle :: B \text{ and } \theta \sqcup \theta' \text{ is consistent and} \\ & T_i = \tau_P^x(\langle B_i :: B \Diamond \theta \sqcup \theta' \sqcup (q = H_i) \rangle) \text{ where } H_i : -B_i \text{ is the } i^{\text{th}} \text{ of} \\ & \text{the } n \text{ clauses of } P \text{ with head } q \text{ (renamed with all new variables)} \\ & \text{and } q = H_i \text{ adds the equality constraints between the actual} \\ & \text{parameters of } q \text{ and formal parameters of } H_i \\ [T_1, \dots, T_n] & \text{if } G = \langle \theta' ? \{ \lambda \}, q \rangle :: B \text{ and } \theta \sqcup \theta' \text{ is consistent and} \\ & T_i = \tau_P^x(\langle B_i :: B \Diamond \theta \sqcup \theta' \sqcup (q = H_i) \rangle) \text{ where } H_i : -B_i \text{ is the } i^{\text{th}} \text{ of} \\ & \text{the } n \text{ clauses of } P \text{ with head } q \text{ (renamed with all new variables)} \\ & \text{and } q = H_i \text{ adds the equality constraints between the actual} \\ & \text{parameters of } q \text{ and formal parameters of } H_i \end{array} \right.$$

Definition 48 [Inclusive Operational Semantics $\tau_P^i : \text{State} \mapsto \text{Tree}$] The inclusive operational semantics of a program P is a mapping $\tau_P^i : \text{State} \mapsto \text{Tree}$ defined as follows:

$$\tau_P^i(\langle G \Diamond \theta \rangle) = \text{tree}(\langle G \Diamond \theta \rangle, L)$$

where

$$L = \left\{ \begin{array}{ll} [] & \text{if } G = \epsilon \\ [] & \text{if } G = \langle \theta', q \rangle :: B \text{ and} \\ & \theta \sqcup \theta' \text{ is inconsistent or no clause in } P \text{ has head } q \\ [] & \text{if } G = \langle \theta' ? \{\lambda\}, q \rangle :: B \text{ and} \\ & \theta \sqcup \theta' \sqcup \{\lambda\} \text{ is inconsistent or no clause in } P \text{ has head } q \\ [T_1, \dots, T_n] & \text{if } G = \langle \theta', q \rangle :: B \text{ and } \theta \sqcup \theta' \text{ is consistent and} \\ & T_i = \tau_P^i(\langle B_i :: B \Diamond \theta \sqcup \theta' \sqcup (q = H_i) \rangle) \text{ where } H_i : -B_i \text{ is the } i^{\text{th}} \text{ of} \\ & \text{the } n \text{ clauses of } P \text{ with head } q \text{ (renamed with all new variables)} \\ & \text{and } q = H_i \text{ adds the equality constraints between the actual} \\ & \text{parameters of } q \text{ and formal parameters of } H_i \\ [T_1, \dots, T_n] & \text{if } G = \langle \theta' ? \{\lambda\}, q \rangle :: B \text{ and } \theta \sqcup \theta' \sqcup \{\lambda\} \text{ is consistent and} \\ & T_i = \tau_P^i(\langle B_i :: B \Diamond \theta \sqcup \theta' \sqcup \{\lambda\} \sqcup (q = H_i) \rangle) \text{ where } H_i : -B_i \text{ is the} \\ & i^{\text{th}} \text{ of the } n \text{ clauses of } P \text{ with head } q \text{ (renamed with new variables)} \\ & \text{and } q = H_i \text{ adds the equality constraints between the actual} \\ & \text{parameters of } q \text{ and formal parameters of } H_i \end{array} \right.$$

Theorem 16 [Failure-Free Reorderings are Admissible] Let R be a program consisting of the sequence of clauses $c_1 \dots c_i \dots c_n$, where the clause c_i is

$$p : - \langle \theta_1, q_1 \rangle, \dots, \langle \theta_k ? \{\lambda\}, q_k \rangle, \langle \theta_{k+1} \sqcup \{\lambda\}, q_{k+1} \rangle, \dots, \langle \theta_m, q_m \rangle.$$

Denote by program point A the point in the clause c_i just before $\langle \theta_k ? \{\lambda\}, q_k \rangle$. Denote by Θ the set of all constraint stores that can occur as the accumulated constraint store at the program point A in the inclusive execution of program R for the query S . Let

$$\forall \theta \in \Theta : \lambda \text{ FF } \tau_R^x(\langle \langle \theta_k ? \{\lambda\}, q_k \rangle \Diamond \theta \rangle).$$

Then

$$\tau_R^i(S) \approx \tau_R^x(S).$$

Proof: Consider a "hybrid" operational semantics for the program R that consists of using the inclusive operational semantics upto depth d and then switching to the exclusive operational semantics. The hybrid operational semantics of program R can be defined formally as a mapping $\tau_R^h : \text{State} \times \text{Integer} \mapsto \text{Tree}$ given by:

$$\tau_R^h(\langle G \Diamond \theta \rangle, d) = \begin{cases} \tau_R^x(\langle G \Diamond \theta \rangle) & \text{if } d = 1 \\ \text{tree}(\langle G \Diamond \theta \rangle, L) & \text{otherwise} \end{cases}$$

where

$$L = \left\{ \begin{array}{ll} [] & \text{if } G = \epsilon \\ [] & \text{if } G = \langle \theta', q \rangle :: B \text{ and} \\ & \theta \sqcup \theta' \text{ is inconsistent or no clause in } R \text{ has head } q \\ [] & \text{if } G = \langle \theta' ? \{\lambda\}, q \rangle :: B \text{ and} \\ & \theta \sqcup \theta' \sqcup \{\lambda\} \text{ is inconsistent or no clause in } R \text{ has head } q \\ [T_1, \dots, T_n] & \text{if } G = \langle \theta', q \rangle :: B \text{ and } \theta \sqcup \theta' \text{ is consistent and} \\ & T_i = \tau_R^h(\langle B_i :: B \Diamond \theta \sqcup \theta' \sqcup (q = H_i) \rangle, d-1) \text{ where } H_i : -B_i \text{ is the} \\ & i^{th} \text{ of the } n \text{ clauses of } R \text{ with head } q \text{ (renamed with new variables)} \\ & \text{and } q = H_i \text{ adds the equality constraints between the actual} \\ & \text{parameters of } q \text{ and formal parameters of } H_i \\ [T_1, \dots, T_n] & \text{if } G = \langle \theta' ? \{\lambda\}, q \rangle :: B \text{ and } \theta \sqcup \theta' \sqcup \{\lambda\} \text{ is consistent and} \\ & T_i = \tau_R^h(\langle B_i :: B \Diamond \theta \sqcup \theta' \sqcup \{\lambda\} \sqcup (q = H_i) \rangle, d-1) \text{ where } H_i : -B_i \\ & \text{is } i^{th} \text{ of the } n \text{ clauses of } R \text{ with head } q \text{ (renamed with new} \\ & \text{variables) and } q = H_i \text{ adds the equality constraints between the} \\ & \text{actual parameters of } q \text{ and formal parameters of } H_i \end{array} \right.$$

Consider the top level query S . Because the hybrid operational semantics mimics the inclusive semantics upto depth d , the hybrid execution tree $\tau_R^h(S, d)$ is identical to the inclusive execution tree $\tau_R^i(S)$ upto depth d . We need to prove that the following invariant holds for any depth d :

$$\tau_R^h(S, d) \approx \tau_R^x(S).$$

It follows trivially that for any depth d :

$$\tau_R^i(S) \approx_d \tau_R^x(S).$$

The proof is by induction on d .

Basis: The basic case is obvious, since

$$\tau_R^h(S, 1) = \tau_R^x(S).$$

Induction Step: Assume that the invariant is true for depth d , i.e.

$$\tau_R^h(S, d) \approx \tau_R^x(S).$$

We show that it remains true for depth $d+1$ by showing that

$$\tau_R^h(S, d+1) \approx \tau_R^h(S, d).$$

The invariant follows for depth $d + 1$ from the transitivity of the similarity relation.

Because $\tau_R^h(S, d)$ and $\tau_R^h(S, d + 1)$ are identical upto depth d , there is a 1-1 correspondence between their nodes upto and including depth d . They may diverge below depth d because $\tau_R^h(S, d)$ uses the exclusive semantics at that point while $\tau_R^h(S, d + 1)$ uses the inclusive semantics for one more level.

Consider a node N at depth d in $\tau_R^h(S, d)$. We show that the corresponding node N' in $\tau_R^h(S, d + 1)$ is either exactly the same as N or it is similar to N . N and N' must have the same labels (because $\tau_R^h(S, d)$ and $\tau_R^h(S, d + 1)$ are identical upto depth d), however their child lists may differ. Let N be $\text{tree}(\langle G \Diamond \theta \rangle, L)$ and N' be $\text{tree}(\langle G \Diamond \theta \rangle, L')$. We need to systematically consider the various cases in the operational semantics. The child list of N is determined by the exclusive operational semantics (Definition 47) while the child list of N' is determined by the hybrid operational semantics.

Case $G = \epsilon$.

Here $L = L' = []$, and so $N = N'$.

Case $G = \langle \theta', q \rangle :: B$ and $\theta \sqcup \theta'$ is inconsistent or no clause in R has head q .

Again, $L = L' = []$, and so $N = N'$.

Case $G = \langle \theta_k ? \{ \lambda \}, q_k \rangle :: B$ and $\theta \sqcup \theta_k$ is inconsistent or no clause in R has head q_k .

We must have that $\theta \sqcup \theta_k \sqcup \{ \lambda \}$ is also inconsistent or no clause in R has head q_k . Hence $L = L' = []$, and so $N = N'$.

Case $G = \langle \theta', q \rangle :: B$ and $\theta \sqcup \theta'$ is consistent and R has a clause with head q .

Let $H_i : -B_i$ ($1 \leq i \leq n$) be the n clauses of R with head q . Here $L = [T_1, \dots, T_n]$ with

$$T_i = \tau_R^x(\langle B_i :: B \Diamond \theta \sqcup \theta' \sqcup (q = H_i) \rangle).$$

Also $L' = [T'_1, \dots, T'_n]$ with

$$T'_i = \tau_R^h(\langle B_i :: B \Diamond \theta \sqcup \theta' \sqcup (q = H_i) \rangle, 1) = \tau_R^x(\langle B_i :: B \Diamond \theta \sqcup \theta' \sqcup (q = H_i) \rangle) = T_i.$$

Here we used the fact that $\tau_R^h(S, d + 1)$ uses the inclusive semantics only for one more level than $\tau_R^h(S, d)$, and so the trees rooted at the children of N' use the exclusive semantics (as they are at depth $d + 1$). We have that $L = L'$, and so $N = N'$.

Case $G = \langle \theta_k ? \{ \lambda \}, q_k \rangle :: B$ and $\theta \sqcup \theta_k$ is consistent and R has a clause with head q_k .

Let $H_i : -B_i$ ($1 \leq i \leq n$) be the n clauses of R with head q . Here $L = [T_1, \dots, T_n]$ with

$$T_i = \tau_R^x(\langle B_i :: B \Diamond \theta \sqcup \theta_k \sqcup (q_k = H_i) \rangle).$$

Since N corresponds to the program point of interest (A), and since $\tau_R^h(S, d)$ is identical to $\tau_R^i(S)$ up to depth d , we have that $\theta \in \Theta$. Hence, by hypothesis,

$$\lambda \text{ FF } \tau_R^x(\langle \langle \theta_k ? \{\lambda\}, q_k \rangle \Diamond \theta \rangle).$$

and, by Lemma 21,

$$\lambda \text{ FF } \tau_R^x(\langle \langle \theta_k ? \{\lambda\}, q_k \rangle :: B \Diamond \theta \rangle).$$

In other words, λ is failure-free in N . By Lemma 19, λ is failure-free for every child T_i of N . This gives us that

$$\forall i : \theta \sqcup \theta_k \sqcup \{\lambda\} \sqcup (q_k = H_i) \text{ is consistent.}$$

Hence $L' = [T'_1, \dots, T'_n]$ where

$$T'_i = \tau_R^h(\langle B_i :: B \Diamond \theta \sqcup \theta_k \sqcup \{\lambda\} \sqcup (q_k = H_i) \rangle, 1) = \tau_R^x(\langle B_i :: B \Diamond \theta \sqcup \theta_k \sqcup \{\lambda\} \sqcup (q_k = H_i) \rangle).$$

Again, we used the fact that $\tau_R^h(S, d+1)$ uses the inclusive semantics only for one more level than $\tau_R^h(S, d)$. By Lemma 20 we have

$$\forall i : T_i \approx T'_i.$$

It follows that $N \approx N'$. This completes the proof. □

Bibliography

- [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, B. Le Charlier, and P. Van Hentenryck. Combinations of Abstract Domains for Logic Programming. In *21st Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, Portland, OR, January 1994.

- [10] 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.
- [11] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Conf. Record of Fifth ACM Symposium on Principles of Programming Languages (POPL'78)*, 1978.
- [12] 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.
- [13] 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.
- [14] V. Dumortier and G. Janssens. Towards a Practical Full Mode Inference System for CLP(H,N). In *Eleventh International Conference on Logic Programming (ICLP-94)*, Santa Margherita Ligure, Italy, June 1994.
- [15] 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.
- [16] M.L. Fredman and R.E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *JACM*, 34:596–615, July 1987.
- [17] 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.
- [18] N. Heintze and J. Jaffar. An Engine for Logic Program Analysis. In *IEEE 7th Annual Symposium on Logic in Computer Science*, 1992.
- [19] M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey Incremental Analysis of Logic Programs. In *1995 International Conference on Logic Programming*, Japan, June, 1995.

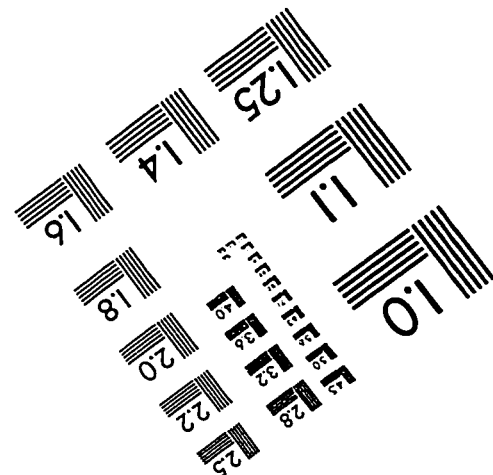
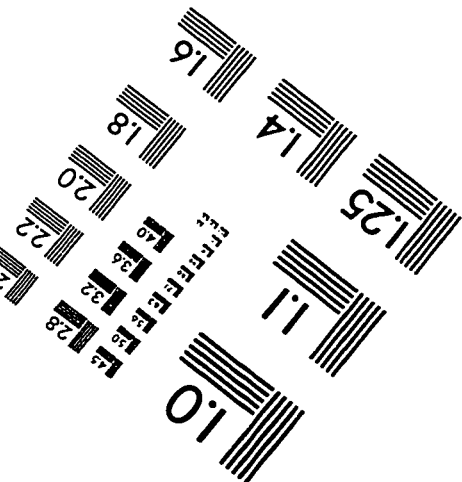
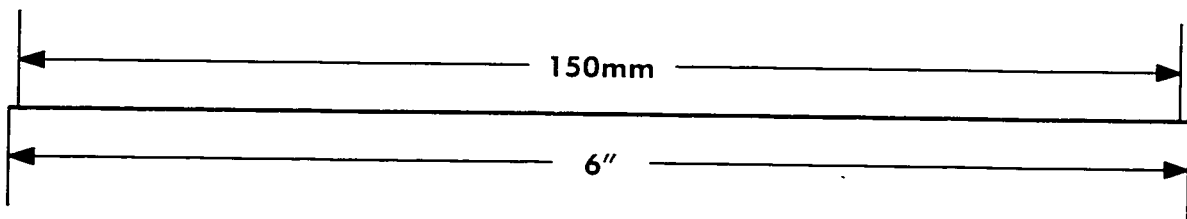
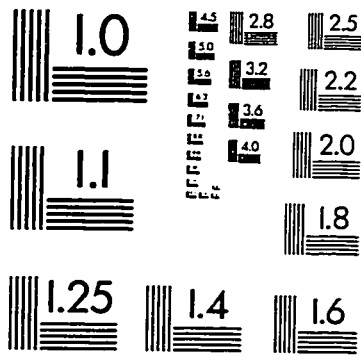
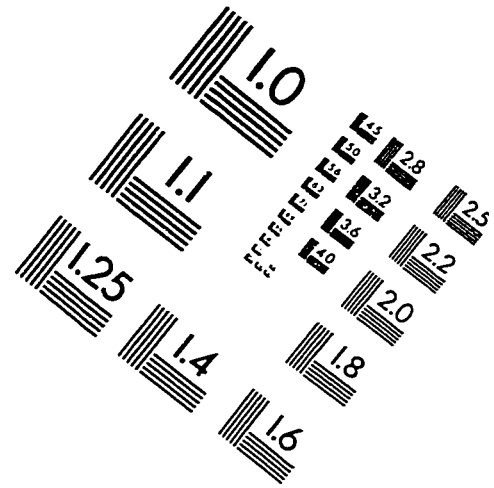
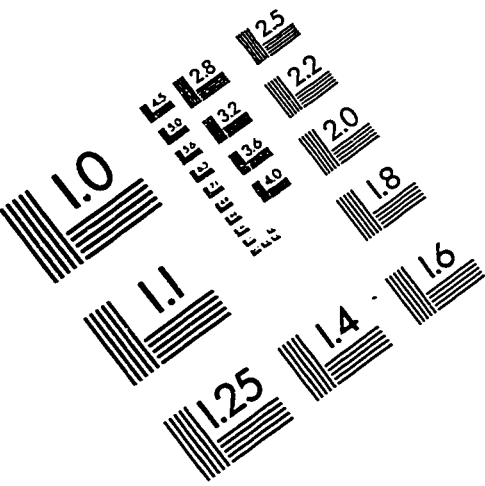
- [20] J.-L. Imbert, J. Cohen, and M.-D. Weeger. An Algorithm for Linear Constraint Solving: Its Incorporation in a Prolog Meta-Interpreter for CLP. *Journal of Logic Programming* 16, 3/4 (July 1993), 235–254.
- [21] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *POPL-87*, (Munich, Germany), January 1987.
- [22] J. Jaffar, S. Michaylov, P.J. Stuckey, and R. Yap. The CLP(\mathbb{R}) language and system. *ACM Trans. on Programming Languages and Systems*, 14(3):339–395, 1992.
- [23] J. Jaffar, S. Michaylov, P.J. Stuckey, and R. Yap. An Abstract Machine for CLP(\mathbb{R}). In *Proc. ACM-SIGPLAN Conf. on Programming Language Design and Implementation (PLDI-92)*, 128–139, 1992, ACM Press.
- [24] N. Jorgensen, K. Marriott, and S. Michaylov. Some Global Compile Time Optimizations for CLP(\mathbb{R}). In *Logic Programming: Proc. 1991 Intl. Symp.*, 420–434, MIT Press, 1991.
- [25] T. Kanamori and T. Kawamura. Analysing Success Patterns of Logic Programs by Abstract Hybrid Interpretation. Technical Report, ICOT, 1987.
- [26] A.D. Kelly, A. MacDonald, K. Marriott, H. Sondergaard, P. Stuckey, and R. Yap. An Optimizing Compiler for CLP(\mathbb{R}). In *First International Conference on Principles and Practice of Constraint Programming (CP'95)*, Cassis, France, September 1995. Springer Verlag.
- [27] A.D. Kelly, A.D. Macdonald, K. Marriott, P.J. Stuckey, and R.H.C. Yap. Effectiveness of Optimizing Compilation for CLP(\mathbb{R}) In Michael J. Maher (Ed.) *Proc. Joint International Conference and Symposium on Logic Programming*, 37–51, MIT Press, 1996.
- [28] 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.
- [29] 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.
- [30] J. W. Lloyd. *Foundations of Logic Programming*, Springer-Verlag, New York, 1984.

- [31] A. MacDonald, P. Stuckey, and R. Yap. Redundancy of Variables in CLP(\mathbb{R}). In *Logic Programming: Proc. 1993 Intl. Symp.*, 75–93, 1993, MIT Press.
- [32] 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.
- [33] K. Marriott, H. Sondergaard and N. D. Jones. Semantics-based Dataflow Analysis of Logic Programs. In *Information Processing-89*, 601–606, San Francisco, CA, 1989.
- [34] K. Marriott and H. Sondergaard. Denotational Abstract Interpretation of Logic Programs, June 1990. *ACM Transaction on Programming Languages and Systems*, 16(3): 607-648, 1994.
- [35] 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.
- [36] 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.
- [37] 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.
- [38] C. Mellish. *Abstract Interpretation of Prolog Programs*, 181–198, Ellis Horwood, Chichester, 1987.
- [39] 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.
- [40] 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.
- [41] R.A. O'Keefe. Finite Fixed-Point Problems. in *Fourth Intl. Conf. on Logic Programming*, 729–743, Melbourne, Australia, 1987.

- [42] W. Older and A. Vellino. Extending Prolog with Constraint Arithmetics on Real Intervals. In *Canadian Conference on Computer & Electrical Engineering*, Ottawa, 1990.
- [43] V. Ramachandran and P. Van Hentenryck. LSign Reordered. In *Static Analysis Symposium (SAS-95)*, Glasgow, UK, September 1995. Lecture Notes in Computer Science, Vol. 983, 330-347, Springer Verlag.
- [44] V. Ramachandran, P. Van Hentenryck, and A. Cortesi. Abstract Domains for Re-ordering of $CLP(\mathcal{R}_{Lin})$. Technical Report, CS-97-07, Brown University, June 1997.
- [45] 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.
- [46] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series, The MIT Press, Cambridge, Mass., 1989.
- [47] 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.
- [48] P. Van Hentenryck and V. Ramachandran. Backtracking without Trailing in $CLP(\mathcal{R}_{Lin})$. In *Proc. ACM-SIGPLAN Conf. on Programming Language Design and Implementation (PLDI-94)*, Orlando, FL, June 1994.
- [49] P. Van Hentenryck and V. Ramachandran. Backtracking without Trailing in $CLP(\mathcal{R}_{Lin})$. *ACM Transactions on Programming Languages and Systems*, July 1995.
- [50] 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.
- [51] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type Analysis of Prolog Using Type Graphs. *Journal of Logic Programming*, 1994.
- [52] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of Prop. *Journal of Logic Programming*, 23(3), June 1995.
- [53] 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.

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

IMAGE EVALUATION TEST TARGET (QA-3)



APPLIED IMAGE, Inc.
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved