Numerical Computation As Deduction In Constraint Logic Programming

by
B.Math., University of Waterloo, Canada, 1987
M.Math., University of Waterloo, Canada, 1988

B.Math., University of Waterloo, Canada, 1987
M.Math., University of Waterloo, Canada, 1988

ACCEPTED
FACULTY OF GRADUATE STUDIES
DEAN
DATE [illegible]

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor in Philosophy
in the Department of Computer Science

We accept this dissertation as conforming
to the required standard

Dr. M.H. van Emden, Supervisor (Department of Computer Science)

Dr. M.H.M. Cheng, Departmental Member (Department of Computer Science)

Dr. M.R. Levy, Departmental Member (Department of Computer Science)

Dr. C.G. Morgan, Outside Member (Department of Philosophy)

Dr. A. Vellino, External Examiner (Computing Research Laboratory, Bell-Northern Research)

©Jimmy Ho Man Lee, 1992
University of Victoria

*All rights reserved. This dissertation may not be reproduced
in whole or in part, by mimeograph or other means,
without the permission of the author.*

Supervisor: Dr. M.H. van Emden

Abstract

Logic programming realizes the ideal of "computation as deduction," except when floating-point arithmetic is involved. In that respect, logic programming languages suffer the same deficiency as conventional algorithmic languages: floating-point operations are only approximate and it is not easy to tell how good the approximation is. This dissertation proposes a framework to extend the benefits of logic programming to computations involving floating-point arithmetic.

John Cleary incorporated a relational form of interval arithmetic into Prolog so that variables already bound can be bound again. In this way, the usual logical interpretation of computation no longer holds. Based on Cleary's idea, we develop a technique for narrowing intervals. We present a relaxation algorithm for coordinating the applications of the interval narrowing operations to constraints in a network.

We incorporate relational interval arithmetic into two constraint logic programming languages: CHIP and CLP($\mathcal{R}$). We modify CHIP by allowing domains to be intervals of real numbers. In CLP($\mathcal{R}$), we represent intervals by inequality constraints. The enhanced languages ICHIP and ICLP($\mathcal{R}$) preserve the semantics of logic so that numerical computations are deductions, even when floating-point arithmetic is used. We have constructed a prototype of ICLP($\mathcal{R}$), consisting of a meta-interpreter executed by an existing CLP($\mathcal{R}$) system.

We show that interval narrowing belongs to the class of domain restriction operations in constraint-satisfaction algorithms. To establish a general framework for these operations, we use and generalize Ashby's notions of cylindrical closure and cylindrance. We show that Mackworth's algorithms can be placed in our framework.

Examiners:

Dr. M.H. van Emden, Supervisor (Department of Computer Science)

Dr. M.H.M. Cheng, Departmental Member (Department of Computer Science)

Dr. M.R. Levy, Departmental Member (Department of Computer Science)

Dr. C.G. Morgan, Outside Member (Department of Philosophy)

Dr. A. Vellino, External Examiner (Computing Research Laboratory, Bell-Northern Research)

Contents

Abstract ii

Contents iv

List of Algorithms viii

List of Figures ix

List of Programs x

List of Tables xi

Acknowledgements xii

Dedication xv

1 Introduction 1

1.1 Computation Should Be Deduction 1

1.2 Is Numerical Computation Deduction? 3

1.2.1 Floating-point Arithmetic 3

1.2.2 Floating-point Arithmetic In Logic Programming 4

1.2.3 Floating-point Arithmetic In Constraint Logic Programming . 6

1.3 Interval Arithmetic Should Be Relational 8

1.4 The Domain Restriction Operations 9

1.5 Related Work 11

1.5.1 Exact Real Arithmetic 12

1.5.2 Interval Arithmetic 12

1.5.3 Constraint Interval Reasoning 13

1.5.4 Constraint Logic Programming 14

1.6 An Overview Of The Dissertation 16

2 Relational Interval Arithmetic **17**

2.1 Basics Of Interval Arithmetic 17

2.2 Outward Rounding With The IEEE Standard 22

2.2.1 Rounding Direction 23

2.2.2 Exceptions 23

2.2.3 Decimal Strings Versus Floating-point Numerals 24

2.3 Interval Narrowing 24

2.4 Arithmetic Primitives 28

2.4.1 Equality 29

2.4.2 Inequalities 29

2.4.3 Addition 29
2.4.4 Multiplication 30
2.4.5 Disequality 33
2.4.6 Transcendental Functions 34
2.5 Constraint Networks 36
2.6 Domain Splitting 42

3 Extending CHIP With Interval Arithmetic 44
3.1 Approximating An Arithmetic Relation 45
3.2 Interval Narrowing As LAIR 46
3.3 Constraint Relaxation 47
3.4 The ICHIP Language 50
3.5 Domain Splitting And Answer Interpretation 50

4 Extending CLP($\mathcal{R}$) With Interval Arithmetic 52
4.1 The Modified CLP Scheme 54
4.2 ICLP($\mathcal{R}$) 55
4.3 An ICLP($\mathcal{R}$) Interpreter 56

5 Analysis Of The Domain Restriction Operations 60
5.1 Basic Set Theory 60
5.2 The Ashby Chain 64
5.3 The Generalized Ashby Chain 67

5.4 The General Scheme 69

5.5 Instances Of The General Scheme 71

6 Concluding Remarks **78**

6.1 Summary And Contributions 78

6.2 Suggestions For Further Work 80

Bibliography **83**

A Consistency Techniques In Logic Programming **93**

A.1 Domain Variables 93

A.2 Consistency Techniques 95

A.3 Using LAIR In CHIP 97

B The CLP Scheme And CLP($\Re$) **98**

B.1 Structure 99

B.2 $\mathcal{M}_{\mathcal{X}}$ Models 101

B.3 Logical Theory 102

B.4 Operational Semantics 103

B.5 CLP($\Re$) 104

List of Algorithms

2.1 A Relaxation Algorithm 39

List of Figures

2.1 The outward-rounding function. 20

2.2 The interval narrowing operation for $(\texttt{le}, \langle I_1, I_2 \rangle)$. 28

2.3 Graphs of some transcendental functions. 35

List of Programs

1.1 A logic program about kinship relationships. 2

1.2 A Prolog program for calculating mortgage information. 5

1.3 A CLP($\Re$) program for calculating mortgage. 7

4.1 A CLP($\mathcal{R}$) meta-interpreter. 57

4.2 The top level of an iCLP($\mathcal{R}$) interpreter in CLP($\mathcal{R}$). 59

List of Tables

2.1 Interval narrowing of $(\texttt{add}, \langle V_1, [1,1], V \rangle)$. 41

2.2 Interval narrowing of $(\texttt{multiply}^+, \langle V, V_1, [6,6] \rangle)$. 41

2.3 Traces of A, P, V, and V_1. 42

2.4 Reduction of an inconsistent network. 43

Acknowledgements

I express the deepest gratitude to my supervisor, Professor Maarten van Emden, who guided me through my study with extreme patience in the last four years. Maarten introduced me to the beauty of logic programming and later encouraged me to get into the fascinating world of constraint logic programming. His insights led to several key ideas in this dissertation. Maarten's insistence on vigor in technical writing helped me to understand that it is equally important for a researcher to present research results as well as to produce them. At the trough of my study, Maarten comforted me and promised to *coach me until the end*. I owe much to Maarten's encouragement. Last but not least, Maarten provided me with adequate financial support when I needed money most.

Dr. André Vellino, my external examiner, gave constructive comments on my thesis and defense. As my meditation mentor, André provided me with encouragement and guidance. His *Everything is Equally Interesting* is more than an article of amusement. André is also generous in sharing his personal philosophy with me. I look at the world around me very differently now. His sincerity and serenity are what I value the most.

I am greatly indebted to Dr. Mantis Cheng, who was instrumental in helping me to start my research. He convinced me to specialize in logic programming and was my personal coach in the area when I was in my Master's program. A discussion with Mantis led to one chapter of this dissertation. His critical comments helped to improve the quality of this dissertation. Mantis is not just my teacher but also my good friend.

I thank my committee members Professor Charles Morgan and Dr. Michael Levy. Professor Charles Morgan helped me to uncover weaknesses in both my research and

writing. Dr. Levy pointed out that applications are important too.

Dr. Paul Strooper has been my senior, my colleague, my friend, and my drinking buddy. He read many of my manuscripts thoroughly, including every chapter of this dissertation, and helped me to improve my writing style.

Discussions with Professor John Cleary, Olivier Lhomme. Bill Older, Professor Stott Parker, Dr. André Vellino, and Dr. Clifford Walinsky have helped to clarify and generated many ideas of my work. Dr. Spiro Michaylov and Roland Yap gave me much help in using CLP($\mathcal{R}$). Their prompt attention of my requests and excellent support made my implementation work almost an enjoyment. I graduated my grade one CLP($\mathcal{R}$) hacking course under Spiro.

I thank all members, past and present, of the Logic Programming Laboratory at UVic: Maarten, Mantis, David, Paul, Rajiv, Brad, Lu, Wayne, Albert, Husain, Panos, Csaba, and Dan. They provided me with a friendly and stimulating research environment.

My appreciation also goes to our mighty system administrator, Will Kastelic, for maintaining a stable computing environment. Will was always there to fulfill my computing needs.

My last four years at UVic were made pleasant by my friends David, Du & Tao, Kim, Mehmet, Michael, Peter, Patrick, Paul, Rajiv, Randal Rod, Xiaoling, and their families. Besides ping pong, soccer, tennis, and outing trips, I shall remember the many coffee breaks, lunches, dinners, and drinking sessions. I must mention that I got to know my wife from Patrick.

My email penpals Amy, Chung, Fiona, Joe, Lhf, Pat, and Sam (my brother) provided me with constant entertainment throughout the years. Their mail has always been a source of enlightenment. I look forward to it everyday.

I must thank my personal numerical analysis consultants: Dr. Eduardo D'Azevedo, Joe, Pat, and Dr. Mark Mutrie. From them, I learned my lessons in error analysis. Dr. Mutrie further supplied me with the background chapter of his doctoral dissertation and directed me to the work of Kahan.

The Mui family gave me the warmest hospitality during my first year in Victoria. My life would have been very dull without the company of Jason and Justin.

I acknowledge the financial support from a University of Victoria Fellowship and the Institute of Robotics and Intelligent Systems.

To Scarlet, with love and admiration

我幼時什麼都不懂，
讀大學時自以為什麼都懂，
畢業後才知什麼都不懂；
中年時又自以為什麼都懂，
晚年才知道原來什麼都不懂。

林語堂

When I was a child, I knew nothing.

When I was in university, I thought I knew everything.

After I graduated, I found out that I knew nothing.

In my middle age, I thought I knew everything again.

I am old now and I finally realize that I actually know nothing.

Lin Yutang

Chapter 1

Introduction

1.1 Computation Should Be Deduction

Mainstream computing holds that programming should be improved by gradual steps, as exemplified by the methods of structured programming and languages such as Pascal and Ada. Revolutionaries such as Patrick Hayes and Robert Kowalski advocated radical change, as embodied in Hayes's motto [27]

$$\text{computation} = \text{controlled deduction.}$$

According to this approach, each program is a sentence in a logical system and every computation step is a valid inference, so that results are logical consequences of program and data. This method allows for a declarative programming paradigm, where programmers specify only *what* (logic) a problem is, without worrying about *how* (control) to solve the problem. The logic and control components of problem-solving are separated, as expressed in the equation [40]

$$\text{Algorithm} = \text{Logic} + \text{Control.}$$

```
parent(john,mary).
parent(mary,joe).
grandparent(X,Z) ←
    parent(X,Y),parent(Y,Z).
```

Program 1.1: A logic program about kinship relationships.

Logic programming [39], as realized by pure Prolog, is an example of this radical alternative in programming languages and method.

A logic program is a set of Horn clauses. Each clause is an assertion of what is true in the programmer's intended interpretation. The commas are read as logical conjunction and the left arrow is logical implication. Program 1.1 is about the parent and grandparent relationships. The first clause specifies that John is a parent of Mary; the second asserts that Mary is a parent of Joe. The last clause is a general specification of the grandparent relationship: individual `X` is a grandparent of `Z` if there exists another individual `Y` such that `X` is a parent of `Y` and `Y` is a parent of `Z`. We can pose queries to a logic program. For example, the query ← `grandparent(G,joe)` asks who the grandparent of Joe is. The inference system used in logic programming is SLD-resolution [72, 4]. In this example, SLD-resolution returns the answer substitution `G=john`. Logic programming is relational. We can also ask who the grandchild of John is by ← `grandparent(john,S)`. In this case, the answer is `S=joe`. *Most importantly, we can conclude by the soundness of SLD-resolution [4] that* `grandparent(john,joe)` *or the fact that John is a grandparent of Joe is a logical consequence of the program 1.1.*

Logic programming, a symbolic computation framework, fulfills Hayes's goal. The question is whether we can extend Hayes's idea to numerical processing involving floating-point arithmetic, in which roundoff errors arise and destroy the correctness

of computation.

1.2 Is Numerical Computation Deduction?

Mainstream numerical programming is based on floating-point arithmetic, inducing roundoff errors. A computation is typically a series of successive approximations, which halts when two consecutive approximations differ by a sufficiently small amount. This amount is then used as the error estimate. Of course sophisticated error analyses can be made to suggest more certain knowledge. But such analyses are usually time-consuming and valid only asymptotically. We quote from a leading floating-point arithmetic expert, Kahan [37]:

> *Error analyses, especially those concerned with roundoff, are so tedious, so much nastier than the calculations they are intended to validate, and so frequently unrewarding, that they should not be inflicted inconsiderately by one man upon another.*

In the following, we first discuss the problems of floating-point arithmetic in computing. Then we illustrate the effect of roundoff errors on the semantics of logic programming languages that use floating-point arithmetic.

1.2.1 Floating-point Arithmetic

Squeezing infinitely many real numbers into a finite number of hardware bits requires an approximate representation. The floating-point system [56, page 19] is such a representation scheme. Although there are infinitely many integers, the results of integer computations can usually be stored in, say, 32 bits. In contrast, given any

fixed number of bits, most calculations with floating-point numbers cannot be exactly represented using that many bits. Therefore, the result of a floating-point computation must often be truncated or rounded to be stored in its finite representation. The resulting ***roundoff error*** is a characteristic feature of floating-point arithmetic. We explain some problems of roundoff errors:

- Accumulation and propagation of roundoff errors can cause the computation result to have little resemblance to the correct answer.
- Floating-point arithmetic does not obey the usual algebraic laws, such as the associative law. Programmers are responsible for the tedious and error-prone task of ensuring the proper ordering of arithmetic operations to minimize roundoff errors.
- Roundoff errors jeopardize the correctness of a test condition, such as an equality test, in a conditional statement. Choosing the wrong branch destroys the correctness of an algorithm.

1.2.2 Floating-point Arithmetic In Logic Programming

Program 1.2 is a numerical example in Prolog. The **mortgage(P,I,MP,B,T)** predicate denotes a 5-ary relation about mortgage payment, where P is the principal of the mortgage in dollars, I is the monthly interest rate, MP is the monthly payment, and B is the remaining balance after T months of payment. The first clause specifies that the remaining balance is the same as the principal P at the beginning of a mortgage term. The second clause recursively specifies that B is the remaining balance of a principal of P dollars after T payments of MP dollars at interest rate of I, if B is the remaining balance of a principal of NP dollars after T-1 payments of MP dollars at

```
mortgage(P,I,MP,P,0).
mortgage(P,I,MP,B,T) ←
    T>0,
    NP is P*(I+1)-MP,
    NT is T-1,
    mortgage(NP,I,MP,B,NT).
```

Program 1.2: A Prolog program for calculating mortgage information.

the same interest rate. The new principal **NP** is the old principal **P** plus the monthly interest and minus the monthly payment.

Suppose the initial principal is 99999 dollars, the monthly interest rate is 0.01, and the monthly payment is 5000 dollars. The query

← **mortgage(99999, 0.01, 5000, B, 10)**

inquires the remaining balance at the end of 10 monthly payments. ALS Prolog Version 1.01 [79] returns the substitution **B=58150.045**[1] on a SUN 3/280. Assuming the same principal and interest rate, we may want to find the monthly payment so that we can pay off the mortgage in 10 months by the query

← **mortgage(99999, 0.01, MP, 0, 10)**.

Most Prolog systems respond no. Current Prolog implementations, based on floating-point arithmetic, have two problems. First, the arithmetic built-ins, such as the **is** predicate, are functional in nature. They are incompatible with the relational paradigm of logic programming. In this example, they ruin the invertibility of the program. Second, roundoff errors destroy the correctness of the computation. The

[1]SICStus Prolog Version 0.7 [12] returns **B=58150.045213392794** on a SUN 3/280.

correct answer to the first query is `B=58150.04521`. Therefore,

`mortgage(99999, 0.01, 5000, 58150.045, 5000)`

is *not* a logical consequence of program 1.2.

The introduction of the CLP scheme and CLP($\Re$) [33] presents a solution to the first problem.

1.2.3 Floating-point Arithmetic In Constraint Logic Programming

Constraint logic programming generalizes logic programming by replacing unification with constraint solving. Prominent projects include Prolog III [17], CAL [1], Trilogy [3, 76], CHIP [22, 74], and the CLP scheme [33]. Most systems either resort to symbolic algebra methods, or restrict arithmetic to rational or integer numbers.

Both Prolog III and CHIP support a relational version of rational arithmetic by restricting constraints to linear equalities, inequalities, and disequalities. Through the finite domain concept, CHIP has a relational form of integer arithmetic. The floating-point arithmetic facilities of CHIP are as rudimentary as those of Prolog. CAL uses the Buchberger algorithm for computing Gröbner bases [10] to solve nonlinear polynomial equations. For linear equalities and inequalities, CAL employs a constraint solver based on the simplex method (see, for example, [64]). Trilogy solves integer constraints by Presburger arithmetic (see, for example, [8]). In the following, we examine CLP($\Re$), an instance of the CLP scheme, whose implementation is based on floating-point arithmetic.

The CLP scheme [33] defines a family of constraint logic programming languages CLP($\mathcal{X}$), parameterized by a domain of computation $\mathcal{X}$. These languages share the

```
mortgage(P,I,MP,P,0).
mortgage(P,I,MP,B,T) ←
    T>0,
    mortgage(P*(I+1)-MP,I,MP,B,T-1).
```

Program 1.3: A CLP($\Re$) program for calculating mortgage.

same semantic properties. CLP($\Re$), where $\Re$ is the domain of finite trees of reals, is intended for solving real numerical constraints. We can replace the **is** predicates in program 1.2 by the equality predicate = to obtain a mortgage program in CLP($\Re$). Program 1.3 is a more succinct version obtained by programming directly in the domain of real numbers. CLP($\Re$) is truly relational. Not only can it handle the failed query in section 1.2.2 (page 5), it also answers more general queries, such as

$$\leftarrow \texttt{mortgage}(99999, 0.01, \texttt{MP}, \texttt{B}, 10).$$

The query questions the relationship between the monthly payment and remaining balance, fixing the principal, the interest rate, and the number of payments.

CLP($\Re$), a theoretical framework, allows relational arithmetic on reals but its implementation CLP($\mathcal{R}$)[2] [35, 36] is obtained by substituting each real number by a single floating-point approximation. CLP($\mathcal{R}$) Version 1.1 [28] responds as follows to the previously discussed queries:

$\leftarrow \texttt{mortgage}(99999, 0.01, 5000, \texttt{B}, 10)$	returns	$\texttt{B} = 58150$
$\leftarrow \texttt{mortgage}(99999, 0.01, \texttt{MP}, 0, 10)$	returns	$\texttt{MP} = 10558.1$
$\leftarrow \texttt{mortgage}(99999, 0.01, \texttt{MP}, \texttt{B}, 10)$	returns	$\texttt{B} = -10.4622 * \texttt{MP} + 110461$

Round-off errors destroy the soundness of each answer and *disqualify CLP($\mathcal{R}$) computations as deductions.*

[2]CLP($\mathcal{R}$) is not a CLP instance but the name of a CLP($\Re$) implementation.

Roundoff errors can also cause the interpreter to answer **no** to a query when an instance of the query is a logical consequence of the program and vice versa. In addition, CLP($\mathcal{R}$) may behave unpredictably when redundant constraints are involved.

1.3 Interval Arithmetic Should Be Relational

Archimedes used inscribing and circumscribing polygons of a circle to obtain an increasing sequence of lower bounds and a decreasing sequence of upper bounds to the number π. Interval methods [54], inspired by this method of Archimedes, represent a radical alternative to floating-point arithmetic in numerical computation. The ideal of interval arithmetic is to be *sure* that the true value is contained in an interval. Iteration is then used to shrink such an interval until it is smaller than a predetermined bound. Here again the goal is *certainty* of knowledge. We quote from Kahan [37]:

> *... no other development in computer systems would assist engineers and others like them to do numerical computations more safely than would the appearance of Interval Arithmetic as universally accessible in Fortran as are double-precision and complex arithmetic.*

We propose to bring together the two radical streams of computing, constraint logic programming and interval methods. Both streams are, in their present form, deficient. Constraint logic programming lacks control of numerical errors. Interval methods rely on conventional algorithmic languages and hence lack computation as deduction. *We show that the two can be combined in such a way that rigorously justified claims can be made about the error in numerical computation using only floating-point arithmetic.* According to this new framework, program 1.3 can respond as follows to the previous queries, depending on the number of significant digits (in

this case 7) used and the output format:

$\leftarrow$ **mortgage**(99999, 0.01, 5000, B, 10)	returns	$B \in (58150.04, 58150.05)$
$\leftarrow$ **mortgage**(99999, 0.01, MP, 0, 10)	returns	$MP \in (10558.10, 10558.11)$
$\leftarrow$ **mortgage**(99999, 0.01, MP, B, 10)	returns	$B = T1 * MP + T2,$
		$T1 \in (-10.46222, -10.46221),$
		$T2 \in (110461.1, 110461.2)$

Our proposal consists of two parts. Traditional interval arithmetic is functional and has been embedded in functional or imperative languages. First, we develop the required relational version by using an interval narrowing operation based on work by Cleary [15], which is implemented in BNR-Prolog [59, 60] and similar to the one used by Sidebottom and Havens [65].

Second, we show how to incorporate relational interval arithmetic into two constraint logic programming languages, CHIP and CLP($\mathcal{R}$). The extended languages preserve the logic programming semantics so that answers are logical consequences of program and data. Therefore, *numerical computation is deduction*. Relational interval arithmetic is a framework for solving a network of arithmetic constraints. Logic programming allows programmers to specify the network in a concise and declarative fashion, leaving the generation and solving of the constraints to the underlying inference engine. This is again in the spirit of "Algorithm = Logic + Control" by Kowalski [40]. Last but not least, we show that Horn logic is a coherent language to express all of constraints, queries, intervals, answers, and variables.

1.4 The Domain Restriction Operations

The relational interval arithmetic we propose is closely related to the work of Fike [23], Waltz [78], and Montanari [52]. Mackworth [46] systematizes their methods in the

general framework of constraint satisfaction algorithms. The ***constraint satisfaction problem*** (CSP) can be briefly stated as follows:

> We are given a set of variables, a domain of possible values for each variable, and a set of constraints. Each constraint is a relation defined over a subset of the variables, limiting the combination of values that the variables in this subset can take. The goal is to find a ***consistent*** assignment of values to the variables so that all the constraints are satisfied simultaneously.

The CSP is NP-complete, for which no efficient solution method is known. Naive approaches to solve a CSP are ***generate-and-test*** and ***backtracking tree search***. In practice, these methods are too slow because of the large search space involved. Constraint satisfaction algorithms resolve the problem by reducing the search space before performing the tree search.

Each constraint satisfaction algorithm consists of a ***domain restriction operation*** and a ***relaxation algorithm***. Given a set of local constraints, the domain restriction operation removes inconsistent values from the domain of variables that appear in the set. The interval narrowing operation (page 24) is a domain restriction operation. The relaxation algorithm coordinates the application of the domain restriction operation on the constraints in a network. Ullman [71] and Hummel and Zucker [30] analyze these relaxation algorithms, based on the concept of minimization of a figure of merit.

We make a connection between the domain restriction operations and the notion of cylindrance of a relation introduced by Ross Ashby [5]. Using this result, we present a general framework for the domain restriction operations. The idea of our general framework is as follows.

Let c be an n-ary relation representing a constraint. Let c' be a superset of c in such a way that it is easier to determine that a partially specified candidate solution does not belong to c'. We generate candidate solutions systematically and test as soon as possible whether they belong to c'. The result is a, hopefully, drastically reduced set of candidate solutions. Sometimes c' is even empty. Savings occur in two ways: first, when a partially specified candidate solution is rejected as not being a member of c', the entire subtree below the candidate is rejected; second, c' is chosen in such a way that it is easy to test for membership. We are concerned with this latter aspect. We call c' the *circumscribing set* of c.

This is the most general description of our approach. At this level of generality it subsumes most pruning techniques. Our contribution is in identifying a certain class of supersets as being particularly suitable: *these are the intersections of cylinders erected on projections of c.* Our method is a general framework rather than a particular method because there is great latitude in the choice of projections on which to erect the cylinders. In fact, some of the methods of our predecessors can be placed in this framework, as we will show by identifying the required choice of projections.

1.5 Related Work

We review other approaches to sound arithmetic and the role of interval arithmetic in the imperative (or procedural) and the relational programming paradigms. We also discuss proposals that incorporate interval arithmetic into logic programming systems.

1.5.1 Exact Real Arithmetic

Besides interval arithmetic, there are other approaches to avoid roundoff errors. Some subsets of reals are finitely representable and closed under the basic arithmetic operations. By *exact real arithmetic*, we refer to arithmetic on one of these subsets. One such subset is the set of rational numbers. Rational arithmetic is available in symbolic computation packages such as Maple [13]. Another subset is the *constructive reals* [6, 9]. Informally, a constructive real is one for which we can compute an arbitrarily precise approximation. We can represent the constructive reals by the associated algorithms and manipulate the algorithms directly [7, 43]. The computation results are exact in that it abstractly represents an exact answer, as with symbolic computation. Another approach is exact arithmetic on continued fractions [77].

Although exact real arithmetic is not susceptible to cumulative round-off errors, its operations are much slower than their floating-point counterparts. Exact real arithmetic has been embedded in functional and imperative programming systems.

1.5.2 Interval Arithmetic

There has been significant progress in interval arithmetic [54] since its inception in the sixties. A good account of interval arithmetic can be found in [2]. Recent trends in interval arithmetic research, pertaining to the imperative programming paradigm, are summarized in [55].

Interval arithmetic contributes methods guaranteeing correctness at the level of a simple arithmetic expression. As embedded in imperative languages, however, interval arithmetic lacks verification of an entire algorithm involving conditional statements and iterations. This difficulty is caused by the mixing of *logic* (what the problem is) and *control* (how the problem is solved) information in the imperative programming

paradigm. For proving correctness of computed results, numerical analysts, such as Kirchner and Kulisch [38, page 37] and Rump [63, page 109], have also discovered the need for and the advantages of declarative and mathematical statements of numerical problems in computer programs. This work shares the goal of logic programming. Thus interval arithmetic is complementary to existing implementations of logic programming.

Bundy [11] recognized the importance of sound arithmetic and implemented a functional interval arithmetic package in Prolog, which is part of a system for checking the conditions of rewrite rules and the solutions to equations in an algebraic manipulation program.

1.5.3 Constraint Interval Reasoning

Research in a relational form of interval arithmetic stems from constraint propagation techniques. A constraint network consists of *nodes*, representing individual parameters having a particular value (known or unknown), connected by *constraints*, which are relations. We attach a *label*, the set of possible values for a node, to each node in the network. Constraint propagation is a process that deduces information from a local group of constraints in a network and propagates the information to the rest of the network. A particular kind of constraint propagation is *label inference*, which uses the constraints to restrict the label. Constraint interval reasoning is a form of label inference, where the labels attached to the nodes of a constraint network are intervals. Davis [18] gives a good survey of this topic.

The following are special-purpose systems that use interval label inference. ENVISION [19] performs qualitative reasoning about the behavior of physical systems over time. TMM [20] is a temporal constraint system that records and reasons with

changes to the world over time. EMPRESS-A [80] is a temporal reasoning system with lazy evaluation for solving scheduling problems. SPAM [50] performs spatial reasoning. These systems are based on consistency techniques [46] that handle static constraint networks. To generate constraints dynamically during execution, the described systems are equipped with programming languages tailored to the application. A typical language of this kind is not coherent, having separate sub-languages to describe the constraints, the queries, the answers, the nodes in the network, and the labels. TP [31] is a scheme of constraint reasoning on interval arithmetic. It only considers closed intervals but has an approximate representation of open intervals. Its language is similar to LISP.

The above systems use interval arithmetic but do not study the effect of outward-rounding in their theoretical basis.

1.5.4 Constraint Logic Programming

There are several proposals to incorporate interval arithmetic into logic programming systems.

Logical Arithmetic. Cleary [15] incorporates "logical arithmetic," a relational version of interval arithmetic, into Prolog. He introduces a new term "interval," which requires an extension of the unification algorithm. Cleary presents several "squeezing" algorithms that reduce arithmetic constraints over intervals. A constraint relaxation cycle coordinates the execution of the squeezing algorithms. However, there is a semantic problem in this approach. Variables bound to intervals, which are terms in the Herbrand universe, are re-bound to smaller intervals. This is not part of resolution, where a variable can be bound only once. It is not clear in what other, if

any, sense this may be a logical inference. Outward-rounding is implemented but its properties are not studied.

BNR Prolog. BNR Prolog [59, 60] provides constraint interval arithmetic, based on Cleary's idea. It handles only closed intervals. Again variables bound to an interval can be re-bound to smaller intervals. It is unclear how standard logic programming semantics can explain the proof procedure of the interval arithmetic component of BNR Prolog.

Echidna. Sidebottom and Havens [66] designed and implemented a version of relational interval arithmetic for the Echidna constraint reasoning system [26]. It is based on hierarchical consistency techniques [48] and can handle union of disjoint intervals. The direct representation of unions of disjoint intervals avoids situations that otherwise require visiting each disjoint interval in turn by backtracking search. However, maintaining and traversing this hierarchical data structure incurs space and time overhead. A formal analysis of the tradeoffs between backtracking and the space and time overhead remains to be conducted. The hierarchical consistency algorithm used in Echidna is *partial* in the sense of PLAIR [74]. This is because (1) outward-rounding computes a larger interval than that specified by the hierarchical consistency algorithm and (2) unions of disjoint intervals are only approximated by a hierarchical data structure. In addition, it is not clear whether Echidna, being a hybrid system of object-oriented and rule-based constraint programming, fits into the constraint logic programming framework. Our method builds on constraint logic programming languages with established semantics.

1.6 An Overview Of The Dissertation

In chapter 2, we describe relational interval arithmetic. We first review the basic concepts of interval arithmetic, followed by an exposition of the interval narrowing operation. We then present a relaxation algorithm that coordinates the application of the interval narrowing operation on individual constraints in a network. In chapters 3 and 4, we show how we can incorporate relational interval arithmetic into the constraint logic programming languages CHIP and CLP($\mathcal{R}$). The enhanced languages preserve the logic programming semantics. In chapter 5, we study the set-theoretic properties of the domain restriction operations in constraint satisfaction algorithms and generalize them based on Ashby's notion of cylindrance. In chapter 6, we summarize our results and contributions, and indicate directions for future research.

We assume the reader is familiar with the standard concepts and terminology of logic programming (see, for example, [45]). A review of consistency techniques in logic programming and the CLP scheme is given in appendices A and B respectively.

Chapter 2

Relational Interval Arithmetic

Cleary [15] describes several specific algorithms to reduce constraints on intervals. These algorithms work under a basic principle: they narrow intervals associated with a constraint by removing values that do not satisfy the constraint. We study the set-theoretic aspect of the algorithms and generalize them for narrowing intervals constrained by any relation p on $\mathbb{R}^n$ satisfying certain criterion. We then discuss interval narrowing for some common arithmetic relations. Interval narrowing is designed for the reduction of a single constraint. Typically, several constraints interact with one another by sharing intervals, resulting in a constraint *network*. We present an algorithm that coordinates the applications of interval narrowing to constraints in a network.

2.1 Basics Of Interval Arithmetic

A good introduction to interval arithmetic can be found in [2]. We use $\mathbb{R}$ to denote the set of real numbers and $\mathbb{F}$ a set of floating-point numbers. For the purpose of

this dissertation, it suffices to assume that $\mathbb{F}$ is any finite subset of $\mathbb{R}$. If $a, b \in \mathbb{F}$ and $a < b$, then a and b are *adjacent* if there does not exist a $c \in \mathbb{F}$ such $a < c < b$. To represent intervals, we use the usual mathematical notations, such as $(1, 2]$. Intuitively, an interval is a segment, possibly infinite, of the real line. To represent intervals without the lower or the upper bounds, we use as bounds the symbols $-\infty$ and $+\infty$ respectively. Note that $-\infty$ and $+\infty$ can only be used with open bounds since they are not members of $\mathbb{R}$. To be precise, we define the set of *real intervals*, $I(\mathbb{R})$, by

$$\begin{aligned} I(\mathbb{R}) \;=\; & \{(a,b] \mid a \in \mathbb{R} \cup \{-\infty\}, b \in \mathbb{R}\} \bigcup \{[a,b) \mid a \in \mathbb{R}, b \in \mathbb{R} \cup \{+\infty\}\} \bigcup \\ & \{[a,b] \mid a, b \in \mathbb{R}\} \bigcup \{(a,b) \mid a \in \mathbb{R} \cup \{-\infty\}, b \in \mathbb{R} \cup \{+\infty\}\}. \end{aligned}$$

In the above definition, "a" is the *lower* bound and "b" is the *upper* bound. "[" and "]" are used to denote *closed* bounds; and "(" and ")" are used to denote *open* bounds.

We distinguish between real intervals and *floating-point intervals*, whose bounds are floating-point numbers. Replacing $\mathbb{R}$ by $\mathbb{F}$ in the definition of $I(\mathbb{R})$, we obtain the definition of floating-point intervals $I(\mathbb{F})$. Note that real and floating-point intervals differ only in the bounds but every interval, real or floating-point, denotes a set of real numbers. For example, $[e, \pi) = \{x \mid e \leq x < \pi\}$, $(-\infty, 4.5] = \{x \mid x \leq 4.5\}$, and $(-\infty, +\infty) = \mathbb{R}$. An interval can also be empty, as in $(4, 1] = \emptyset$. We impose a partial ordering on real intervals; an interval I_1 is *smaller than or equal to* an interval I_2 if and only if $I_1 \subseteq I_2$. Let $\mathcal{I}$ be a set of intervals ($\mathcal{I} \subseteq I(\mathbb{R})$ or $\mathcal{I} \subseteq I(\mathbb{F})$). $I \in \mathcal{I}$ is the *smallest* interval in $\mathcal{I}$ if I is smaller than or equal to I' for all $I' \in \mathcal{I}$.

If $\cdot \in \{+, -, \times, /\}$, we denote the corresponding real (or floating-point) interval operation by $\odot$:

$$A \odot B = \{a \cdot b \mid a \in A, b \in B\},$$

where A and B are real (or floating-point) intervals. In the case of interval division, $\oslash$, B cannot contain 0. From the above definition, it may not be obvious that $A \odot B$ is always an interval. The following definition and proposition justify the definition.

Definition 2.1.1 [62, page 42]
Two subsets A and B of a metric space X are said to be *separated* if no point of A lies in the closure of B and no point of B lies in the closure of A. A set $E \subset X$ is said to be *connected* if E is *not* a union of two non-empty separated sets.

Proposition 2.1.2 [62, page 93]
If f is a continuous mapping of a metric space X into a metric space Y, and if E is a connected subset of X, then $f(E)$ is connected. ∎

The arithmetic operations $+$, $-$, and $\times$ are continuous. The division operation, $/$, is also continuous if the domain of the denominator does not contain zero. $\mathbb{R}$ is a metric space and a real interval is a connected subset of $\mathbb{R}$. By proposition 2.1.2, $A \odot B$ is guaranteed to be an interval Similarly, we can construct new interval operators based on other real arithmetic operators.

Floating-point intervals are not closed under interval operations. In floating-point arithmetic, real numbers are approximated by floating-point numbers using rounding or truncation. In interval arithmetic, we approximate real intervals by floating-point intervals using the *outward-rounding* function, $\xi : I(\mathbb{R}) \to I(\mathbb{F})$; if J is a non-empty real interval,

$$\xi(J) = \bigcap\{J' \in I(\mathbb{F}) \mid J \subseteq J'\}.$$

The introduction of outward-rounding complicates the proofs of some seemingly simple facts about relational interval arithmetic.

Lemma 2.1.3

If I is a non-empty real interval, then $\xi(I)$ is the smallest floating-point interval containing I.

Proof: Floating-point intervals are closed under the set intersection operation. Therefore, $\xi(I)$ is a floating-point interval. In addition, $I \subseteq \xi(I)$ since $\xi(I)$ is the intersection of all floating-point intervals containing I. Also, if $I \subseteq I'$ for some $I' \in I(\mathbb{F})$, then $\xi(I) \subseteq I'$ by the definition of ξ. ■

Lemma 2.1.3 can be the basis of an implementation of ξ. This is illustrated by the example in figure 2.1. $\mathbb{R}$ is a continuum and $\mathbb{F}$ consists of floating-point numbers

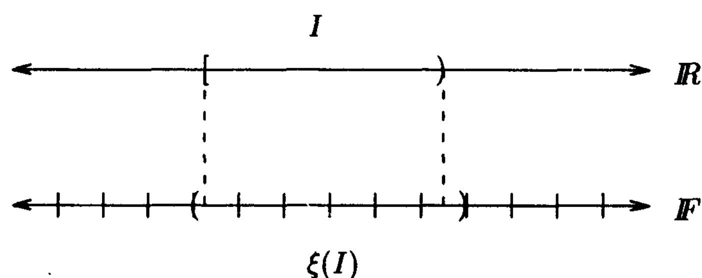

Figure 2.1: The outward-rounding function.

scattered along the real line. Intuitively, we round to the "left" at the lower bound and to the "right" at the upper bound of I.

Lemma 2.1.3 implies that $\xi(I) = I$ for all floating-point intervals I. Thus, ξ is *idempotent*, i.e. $\xi(\xi(I)) = \xi(I)$ for any interval I. In the following, we discuss some properties of ξ. The next lemma shows that outward-rounding is monotone.

Lemma 2.1.4

Let $I_1, I_2 \in I(\mathbb{R})$. If $I_1 \subseteq I_2$, then $\xi(I_1) \subseteq \xi(I_2)$.

Proof: By lemma 2.1.3, $I_1 \subseteq \xi(I_1)$ and $I_1 \subseteq I_2 \subseteq \xi(I_2)$. Since $\xi(I_1)$ is the smallest floating-point interval containing I_1 by lemma 2.1.3, $\xi(I_1) \subseteq \xi(I_2)$. ■

The following results show that every element of $\xi(I)$ is contained in the neighborhood of I.

Lemma 2.1.5
If $I \in I(\mathbb{R})$, then $\xi(I) = \bigcup\{\xi([x,x]) \mid x \in I\}$.

Proof: Let $I' = \bigcup\{\xi([x,x]) \mid x \in I\}$. Suppose $x \in I$. Thus $[x,x] \subseteq I$. By lemma 2.1.4 (the monotonicity of ξ), $\xi([x,x]) \subseteq \xi(I)$. Therefore, $I' \subseteq \xi(I)$.

By the definition of ξ, $[x,x] \subseteq \xi([x,x])$. This implies that $\bigcup\{[x,x] \mid x \in I\} = I \subseteq I'$. We partition $\xi(I)$ as follows: $\xi(I) = L \cup I \cup U$, where L is the extension of I at the lower bound and U is the extension of I at the upper bound. Let a be the lower bound and b the upper bound of I. If $a \in \mathbb{F}$, then $L = \emptyset$ since a is a valid floating-point interval bound. Suppose $a \notin \mathbb{F}$. There exist $m, n \in \mathbb{F}$ such that $m < a < n$, where m and n are adjacent in $\mathbb{F}$. Thus $\xi([a,a]) = (m,n)$. If a is a closed bound, then $L \subseteq (m,n) = \xi([a,a]) \subseteq I'$ since $a \in I$. If a is an open bound, then there exists $a' \in I$ such that $m < a < a' < n$. Thus $L \subseteq (m,n) = \xi([a',a']) \subseteq I'$ since $a' \in I$. We can apply a similar argument at the upper bound b to show that $U \subseteq I'$. Therefore, $\xi(I) \subseteq I'$. ■

Corollary 2.1.6
If $I \in I(\mathbb{F})$ and $x \in I$, then $\xi([x,x]) \subseteq \xi(I) = I$.

Proof: From lemma 2.1.5. ■

Corollary 2.1.7

If $I \in I(\mathbb{F})$, $x \in I$, and $x \in \xi([x', x'])$ for some $x' \in \mathbb{R}$, then $\xi([x', x']) \subseteq \xi(I) = I$.

Proof: If $x' \in \mathbb{F}$, then $\xi([x', x']) = [x', x'] = \{x'\}$ and thus $x = x'$. By corollary 2.1.6, the result holds.

Suppose $x' \notin \mathbb{F}$. Therefore, $\xi([x', x']) = (a, b)$, where a and b are adjacent in $\mathbb{F}$ and $a < b$. Since $x \in (a, b)$, $\xi([x, x]) = (a, b)$. Thus, $x' \in (a, b) = \xi([x, x]) \subseteq \xi(I) = I$. By corollary 2.1.6, the result holds. ∎

In general, ξ does not distribute over set intersection. If I_1 and I_2 are real intervals, then $\xi(I_1 \bigcap I_2)$ may not be equal to $\xi(I_1) \bigcap \xi(I_2)$. For example, $\xi([0, 1/3) \bigcap (1/3, 4])$ is the empty set but $\xi([0, 1/3)) \bigcap \xi((1/3, 4])$ contains at least $1/3$. We have the following relationship for the distribution of ξ over set intersection.

Lemma 2.1.8
If $I_1, I_2 \in I(\mathbb{R})$, then $\xi(I_1 \bigcap I_2) \subseteq \xi(I_1) \bigcap \xi(I_2)$.

Proof: We have $I_1 \cap I_2 \subseteq I_1$. By lemma 2.1.4, $\xi(I_1 \cap I_2) \subseteq \xi(I_1)$. Similarly, $\xi(I_1 \cap I_2) \subseteq \xi(I_2)$. Thus, $\xi(I_1 \bigcap I_2) \subseteq \xi(I_1) \bigcap \xi(I_2)$. ∎

2.2 Outward Rounding With The IEEE Standard

Outward-rounding does not introduce roundoff errors and guarantees that no answer, if there is any, escapes from the rounded interval. This contributes to the guaranteed accuracy property of interval arithmetic. Therefore, a sound implementation of outward-rounding is essential in an interval arithmetic system. In this section, we discuss the implementation of outward-rounding on hardware that conforms to the IEEE floating-point standard. The operations that we shall describe amount to set-

ting or examining a hardware flag before or after a floating-point operation. These capabilities are available in most programming language libraries.

There are two IEEE standards on the implementation of floating-point number systems: one for binary floating-point arithmetic [57] and the other one for radix-independent floating-point arithmetic [58]. The latter is a generalization of the first. For our discussion, it is sufficient to refer to them as "The Standards."

The Standards define a family of commercially feasible ways for new systems to perform floating-point arithmetic. We do not address all aspects of the Standards. Readers are referred to the original documents for further details. We are interested in the parts of the Standards that provide direct support for interval arithmetic: rounding direction, exception handling, and conversions between floating-point numbers and decimal strings.

2.2.1 Rounding Direction

The Standards support four user-selectable *directed rounding modes*: **nearest** (round to nearest), **tozero** (round toward 0), **negative** (round toward $-\infty$), and **positive** (round toward $+\infty$). Each mode is self-explanatory. **Nearest** is the default rounding mode. For the implementation of outward-rounding, we use the **negative** mode to round to the left at the lower bound and the **positive** mode to round to the right at the upper bound.

2.2.2 Exceptions

It is important to monitor exception conditions during floating-point computations. In particular, we are interested in when outward-rounding takes place. There are five

types of exceptions: `invalid`, `inexact`, `division`, `underflow`, and `overflow`. In particular, the `inexact` exception signals the occurrence of outward-rounding.

2.2.3 Decimal Strings Versus Floating-point Numerals

Non-machine arithmetic is usually in the decimal (base 10) system but most digital computers use binary (base 2) numerals. In general, it is impossible to convert numerals between the decimal representations and the binary representations exactly. In that case, the conversion should be rounded correctly. The Standards specifies this conversion procedure under different directed rounding modes.

2.3 Interval Narrowing

An *interval constraint* is of the form $(p, \vec{I})$, where p is a relation on $\mathbb{R}^n$ and $\vec{I} = \langle I_1, \ldots, I_n \rangle$ is a tuple of floating-point intervals, where $I_i \in I(\mathbb{F})$. The interval constraint $(p, \vec{I})$ states that

$$\exists X_i \in I_i (i = 1, \ldots, n) \text{ such that } p(X_1, \ldots, X_n) \text{ holds.}$$

Note that the number of intervals in the tuple $\vec{I}$ is equal to the arity of p. For example, the constraint $(\mathbf{add}, \langle [1,3], [2,4], [4,6] \rangle)$ means that

$$\exists X \in [1,3], \exists Y \in [2,4], \exists Z \in [4,6] \text{ such that } \mathbf{add}(X, Y, Z) \text{ holds.}$$

Given any n-ary relation p, we can determine the possible values of the ith argument of p if we know the possible values of the other arguments. We can thus

associate n set-valued functions with p:

$$\begin{aligned} & F_i(p)(S_1,\ldots,S_{i-1},S_{i+1},\ldots,S_n) \\ = & \{s_i \mid (s_1,\ldots,s_n) \in ((S_1 \times \cdots \times S_{i-1} \times \pi_i(p) \times S_{i+1} \times \cdots \times S_n) \bigcap p)\} \\ = & \pi_i((S_1 \times \cdots \times S_{i-1} \times \pi_i(p) \times S_{i+1} \times \cdots \times S_n) \bigcap p), \end{aligned}$$

where $i = 1,\ldots,n$, the S_i's are sets[1], and π_i is the projection function defined by

$$\pi_i(p) = \{s_i \mid (s_1,\ldots,s_n) \in p\}.$$

Essentially, $F_i(p)(S_1,\ldots,S_{i-1},S_{i+1},\ldots,S_n)$ is the set of possible values for the ith argument of p if we restrict the values of the other arguments to be elements of $S_1,\ldots,S_{i-1},S_{i+1},\ldots,S_n$ respectively. We use $F_i(p)$ as the basis of interval functions. In interval arithmetic, it is essential that the results of interval operations are intervals. Therefore, we consider only relations p on $\mathbb{R}^n$ such that $F_i(p)$ *maps intervals to intervals for* $i = 1,\ldots,n$. For example, the relation $\mathsf{add} = \{(x,y,z) \mid x,y,z \in \mathbb{R}, x + y = z\}$ satisfies this requirement. We have

$$F_1(\mathsf{add})(I_2,I_3) = I_3 \ominus I_2, \qquad F_2(\mathsf{add})(I_1,I_3) = I_3 \ominus I_1, \qquad F_3(\mathsf{add})(I_1,I_2) = I_1 \oplus I_2,$$

where $A \oplus B = \{a + b \mid a \in A, b \in B\}$ and $A \ominus B = \{a - b \mid a \in A, b \in B\}$. By the continuity of the $+$ and $-$ operators, $A \oplus B$ and $A \ominus B$ are intervals if A and B are intervals.

We now specify *interval narrowing* as an input-output pair.

Input: $(p, \langle I_1,\ldots,I_n\rangle)$, where $I_i \in I(\mathbb{F})$ and $p \subseteq \mathbb{R}^n$.

Output: $\vec{I}' = \langle I'_1,\ldots,I'_n\rangle$, where $I'_i = I_i \bigcap \xi(F_i(p)(I_1,\ldots,I_{i-1},I_{i+1},\ldots,I_n))$.

The application of ξ in the formula ensures that the output intervals are floating-point intervals. If one or more I'_i is empty, then interval narrowing *fails* and the constraint

[1]In the present application, S_i's are intervals.

$(p, \vec{I})$ is *inconsistent*. Otherwise it succeeds with $I'_1, \ldots, I'_n$ as output. Note that the output interval I'_i is a subset of the corresponding input interval I_i.

It is important that interval narrowing does not eliminate values that can satisfy the constraint. The following theorem guarantees this property.

Theorem ..3.1
Let C be $(p, \langle I_1, \ldots, I_n \rangle)$ and $\langle I'_1, \ldots, I'_n \rangle$ the output intervals obtained from interval narrowing of C.

$\forall (x_1, \ldots, x_n) \in p : (x_1, \ldots, x_n) \in I_1 \times \cdots \times I_n$ if and only if $(x_1, \ldots, x_n) \in I'_1 \times \cdots \times I'_n$.

Proof: Since $I'_1 \times \cdots \times I'_n \subseteq I_1 \times \cdots \times I_n$, the if-part of the lemma is true. We prove the only-if-part of the lemma. Suppose $(x_1, \ldots, x_n) \in I_1 \times \cdots \times I_n \cap p$. We have $x_i \in I_i$ for $i = 1, \ldots, n$ and $x_i \in F_i(p)(I_1, \ldots, I_{i-1}, I_{i+1}, \ldots, I_n)$ by definition. Therefore $x_i \in I'_i$ and $(x_1, \ldots, x_n) \in I'_1 \times \cdots \times I'_n$. ∎

We can strengthen lemma 2.1.8 if one of the intersecting intervals belongs to $I(\mathbb{F})$ since outward-rounding does not change a floating-point bound.

Lemma 2.3.2
If $I_1 \in I(\mathbb{F})$ and $I_2 \in I(\mathbb{R})$, then $I_1 \bigcap \xi(I_2) = \xi(I_1 \bigcap I_2)$.

Proof: If $I_1 \in I(\mathbb{F})$, then $\xi(I_1) = I_1$. By lemma 2.1.8, $\xi(I_1 \bigcap I_2) \subseteq I_1 \bigcap \xi(I_2)$. Next, we prove that $I_1 \bigcap \xi(I_2) \subseteq \xi(I_1 \bigcap I_2)$.

Suppose $x' \in I_1 \bigcap \xi(I_2)$. Thus $x' \in I_1$ and $x' \in \xi(I_2)$. By lemma 2.1.5, there exists $x \in I_2$ such that $x' \in \xi([x, x])$. By corollary 2.1.7, $x \in I_1$ and thus $x \in \xi(I_1 \bigcap I_2)$. It follows from corollary 2.1.6 that $x' \in \xi(I_1 \bigcap I_2)$. ∎

Lemma 2.3.2 assists in expressing interval narrowing in terms of intersection and projection.

Theorem 2.3.3

$I'_i = \xi(\pi_i((I_1 \times \cdots \times I_n) \bigcap p))$.

Proof:

$$\begin{aligned} I'_i &= I_i \bigcap \xi(F_i(p)(I_1, \ldots, I_{i-1}, I_{i+1}, \ldots, I_n)) \\ &= \xi(I_i \bigcap F_i(p)(I_1, \ldots, I_{i-1}, I_{i+1}, \ldots, I_n)) \text{ by lemma 2.3.2} \\ &= \xi(I_i \bigcap \pi_i((I_1 \times \cdots \times I_{i-1} \times \pi_i(p) \times I_{i+1} \times \cdots \times I_n) \bigcap p) \\ &= \xi(I_i \bigcap \{x_i \,|\, (x_1, \ldots, x_n) \in ((I_1 \times \cdots \times I_{i-1} \times \pi_i(p) \times I_{i+1} \times \cdots \times I_n) \bigcap p))\} \\ &= \xi(\{x_i \in I_i \,|\, (x_1, \ldots, x_n) \in ((I_1 \times \cdots \times I_{i-1} \times \pi_i(p) \times I_{i+1} \times \cdots \times I_n) \bigcap p))\} \\ &= \xi(\{x_i \,|\, (x_1, \ldots, x_n) \in ((I_1 \times \cdots \times I_n) \bigcap p)\}) \\ &= \xi(\pi_i((I_1 \times \cdots \times I_n) \bigcap p)) \end{aligned}$$

■

Theorem 2.3.3 is an alternative definition of interval narrowing. It allows us to understand interval narrowing in terms of relational operations. In essence, interval narrowing computes the intersection of $I_1 \times \cdots \times I_n$ and p, and outward-rounds each projection of the resulting relation. Figure 2.2 illustrates the interval narrowing of the constraint $(\texttt{le}, \langle I_1, I_2 \rangle)$, where $\texttt{le} = \{(x, y) \,|\, x, y \in I\!R, x \leq y\}$. In the diagram, the initial floating-point intervals are I_1 and I_2. The dotted region denotes the relation `le`; the region for $I_1 \times I_2$ is shaded with a straight-line pattern. Interval narrowing returns I'_1 and I'_2 by taking the projections of the intersection of the two regions. There is no need to perform outward-rounding in this example since the bounds of I'_1 and I'_2 share those of I_1 and I_2.

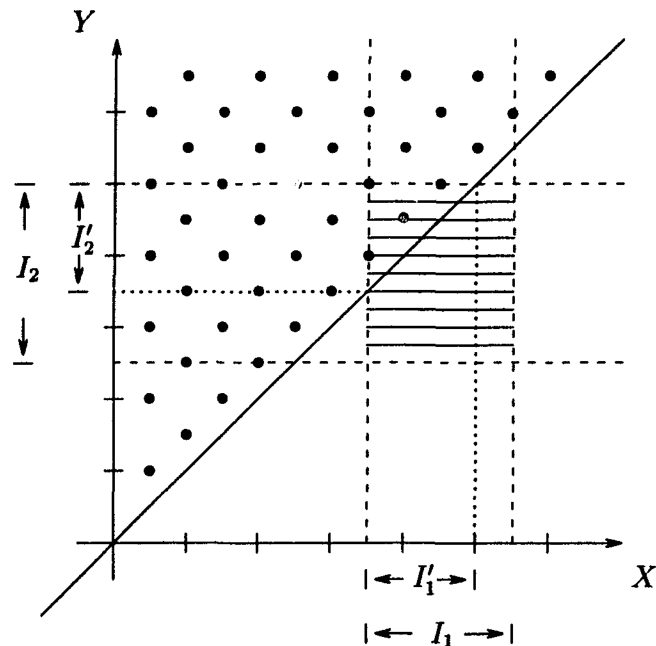

Figure 2.2: The interval narrowing operation for $(\texttt{le}, \langle I_1, I_2 \rangle)$.

2.4 Arithmetic Primitives

A useful relational interval arithmetic system should support some primitive arithmetic constraints, such as addition and multiplication. More complicated constraints can then be built from these primitives. To make a relation p into a primitive, we need to know how to compute each function $F_i(p)$ associated with p. To ensure that p is suitable for interval narrowing, we need to check that each $F_i(p)$ maps from real intervals to real intervals.

2.4.1 Equality

$$\mathsf{eq} = \{(x, x) \mid x \in \mathbb{R}\}$$

The functions $F_1(\mathsf{eq})$ and $F_2(\mathsf{eq})$ are the identity functions on real intervals,

$$F_1(\mathsf{eq})(I) = F_2(\mathsf{eq})(I) = I, \text{ for all } I \in I(\mathbb{R}).$$

Clearly, $F_1(\mathsf{eq})$ and $F_2(\mathsf{eq})$ map from real interval to real interval.

2.4.2 Inequalities

$$\mathtt{le} = \{(x, y) \mid x, y \in \mathbb{R}, x \leq y\} \qquad \mathtt{lt} = \{(x, y) \mid x, y \in \mathbb{R}, x < y\}$$

The functions $F_1(\mathtt{le})$ and $F_2(\mathtt{le})$ are:

$$F_1(\mathtt{le})(I_2) = \begin{cases} (-\infty, b] & \text{if } I_2 = [a, b] \text{ or } (a, b] \\ (-\infty, b) & \text{if } I_2 = (a, b) \text{ or } [a, b) \end{cases}$$

$$F_2(\mathtt{le})(I_1) = \begin{cases} [a, +\infty) & \text{if } I_1 = [a, b] \text{ or } [a, b) \\ (a, +\infty) & \text{if } I_1 = (a, b] \text{ or } (a, b). \end{cases}$$

From the definitions, it is obvious that $F_1(\mathtt{le})$ and $F_2(\mathtt{le})$ map from real interval to real interval. Similarly, we have for the `lt` relation:

$$F_1(\mathtt{lt})(I_2) = (-\infty, b) \quad \text{if } I_2 = [a, b] \text{ or } [a, b) \text{ or } (a, b] \text{ or } (a, b)$$

$$F_2(\mathtt{lt})(I_1) = (a, +\infty) \quad \text{if } I_1 = [a, b] \text{ or } [a, b) \text{ or } (a, b] \text{ or } (a, b).$$

2.4.3 Addition

$$\mathsf{add} = \{(x, y, z) \mid x, y, z \in \mathbb{R}, x + y = z\}$$

If $I_1, I_2, I_3 \in I(\mathbb{R})$, then

$$\begin{aligned} F_1(\mathtt{add})(I_2, I_3) &= I_3 \ominus I_2, \\ F_2(\mathtt{add})(I_1, I_3) &= I_3 \ominus I_1, \\ F_3(\mathtt{add})(I_1, I_2) &= I_1 \oplus I_2, \end{aligned}$$

where

$$\begin{aligned} I_1 \oplus I_2 &= \{a + b \,|\, a \in I_1, b \in I_2\}, \\ I_1 \ominus I_2 &= \{a - b \,|\, a \in I_1, b \in I_2\}. \end{aligned}$$

As mentioned earlier, the + and − operators are continuous. By proposition 2.1.2, we have that $\oplus$ and $\ominus$ map from real intervals to real interval.

2.4.4 Multiplication

$$\mathtt{multiply} = \{(x, y, z) \,|\, x, y, z \in \mathbb{R}, xy = z\}$$

The `multiply` relation involves both multiplication and division. Therefore, we need to use the interval division operation $\oslash$ to define $F_1(\mathtt{multiply})$ and $F_2(\mathtt{multiply})$, where

$$I_1 \oslash I_2 = \{a/b \,|\, a \in I_1, b \in I_2, b \neq 0\} \qquad \text{for } I_1, I_2 \in I(\mathbb{R}).$$

Note that $I_1 \oslash I_2$ is not an interval in general, since division is not continuous when the divisor is 0. For example,

$$[1, 1] \oslash [-2, 3] = (-\infty, -1/2] \cup [1/3, +\infty)$$

is a union of two disjoint intervals. The `multiply` relation does not satisfy the criterion for interval narrowing.

As suggested in [15], we can circumvent the situation by partitioning `multiply` into `multiply`$^+$ and `multiply`$^-$, where

$$\begin{aligned}\texttt{multiply}^+ &= \{(x,y,z) \mid x,y,z \in \mathbb{R}, x \geq 0, xy = z\},\\ \texttt{multiply}^- &= \{(x,y,z) \mid x,y,z \in \mathbb{R}, x < 0, xy = z\}.\end{aligned}$$

By restricting interval narrowing to each partition, we can guarantee that the result of interval division is an interval. When a `multiply` constraint is encountered, we choose one of the partitions and perform the interval narrowing operation; the other partition is visited upon backtracking or under user control.

Before we discuss the functions $F_i(\texttt{multiply}^+)$ and $F_i(\texttt{multiply}^-)$, we define the sets:

$$\begin{aligned}H^+ &= \{x \in H \mid x \geq 0\}, \text{ for any } H \subseteq \mathbb{R}\\ H^- &= \{x \in H \mid x < 0\}, \text{ for any } H \subseteq \mathbb{R}\\ I(\mathbb{R}^+) &= \{(a,b] \mid a \in \mathbb{R}^+, b \in \mathbb{R}^+\} \bigcup\\ &\quad \{[a,b) \mid a \in \mathbb{R}^+, b \in \mathbb{R}^+ \cup \{+\infty\}\} \bigcup\\ &\quad \{[a,b] \mid a,b \in \mathbb{R}^+\} \bigcup \{(a,b) \mid a \in \mathbb{R}^+, b \in \mathbb{R}^+ \cup \{+\infty\}\}\\ I(\mathbb{R}^-) &= \{(a,b] \mid a \in \mathbb{R}^- \cup \{-\infty\}, b \in \mathbb{R}^-\} \bigcup\\ &\quad \{[a,b) \mid a \in \mathbb{R}^-, b \in \mathbb{R}^- \cup \{0\}\} \bigcup\\ &\quad \{[a,b] \mid a,b \in \mathbb{R}^-\} \bigcup \{(a,b) \mid a \in \mathbb{R}^- \cup \{-\infty\}, b \in \mathbb{R}^- \cup \{0\}\}\end{aligned}$$

For `multiply`$^+$ and $I_1, I_2, I_3 \in I(\mathbb{R})$, we have

$$\begin{aligned}F_1(\texttt{multiply}^+)(I_2, I_3) &= I_3 \oslash_1 I_2,\\ F_2(\texttt{multiply}^+)(I_1, I_3) &= I_3 \oslash_2 (I_1 \bigcap \mathbb{R}^+),\\ F_3(\texttt{multiply}^+)(I_1, I_2) &= (I_1 \bigcap \mathbb{R}^+) \otimes I_2,\end{aligned}$$

where

$$A \oslash_1 B = \begin{cases} \mathbb{R}^+ & \text{if } 0 \in A, 0 \in B \\ (A \oslash B) \bigcap \mathbb{R}^+ & \text{otherwise} \end{cases},$$

$$A \oslash_2 C = \begin{cases} \mathbb{R} & \text{if } 0 \in A, 0 \in C \\ A \oslash C & \text{otherwise} \end{cases},$$

$$A \otimes B = \{ab \,|\, a \in A, b \in B\},$$

for $A, B \in I(\mathbb{R})$ and $C \in I(\mathbb{R}^+)$.

For `multiply`$^-$ and $I_1, I_2, I_3 \in I(\mathbb{R})$, we have

$$\begin{aligned} F_1(\texttt{multiply}^-)(I_2, I_3) &= I_3 \oslash_3 I_2, \\ F_2(\texttt{multiply}^-)(I_1, I_3) &= I_3 \oslash_4 (I_1 \bigcap \mathbb{R}^-), \\ F_3(\texttt{multiply}^-)(I_1, I_2) &= (I_1 \bigcap \mathbb{R}^-) \otimes I_2, \end{aligned}$$

where

$$A \oslash_3 B = \begin{cases} \mathbb{R}^- & \text{if } 0 \in A, 0 \in B \\ (A \oslash B) \bigcap \mathbb{R}^- & \text{otherwise} \end{cases},$$

$$A \oslash_4 C = A \oslash C,$$

for $A, B \in I(\mathbb{R})$ and $C \in I(\mathbb{R}^-)$.

It is easy to check that $A \otimes B$ and $A \oslash_4 B$ are real intervals since multiplication and division are continuous, provided that the domain of the divisor does not contain 0 in the case of division. The following lemma shows that $\oslash_1$, $\oslash_2$, and $\oslash_3$ also satisfy the criteria.

Lemma 2.4.1
If $A, B \in I(\mathbb{R})$ and $C \in I(\mathbb{R}^+)$, then

(1) $A \oslash_1 B \in I(\mathbb{R}^+) \subset I(\mathbb{R})$, (2) $A \oslash_2 C \in I(\mathbb{R})$, (3) $A \oslash_3 B \in I(\mathbb{R}^-) \subset I(\mathbb{R})$.

Proof: We will prove (1); the proofs for (2) and (3) are similar.

$\oslash_1$ is defined in terms of real division. When $0 \notin B$, real division is continuous and (1) holds. Now we consider the cases when $0 \in B$.

If $0 \in A$, then

$$A \oslash_1 B = \mathbb{R}^+ \in I(\mathbb{R}^+) \subset I(\mathbb{R}),$$

and thus (1) holds. If $0 \notin A$, there are two cases to consider:

$$\forall(a \in A), a > 0 \text{ or } \forall(a \in A), a < 0.$$

In the first case,

$$\begin{aligned} A \oslash_1 B &= (A \oslash B) \bigcap \mathbb{R}^+ \\ &= ((A \oslash B^-) \bigcup (A \oslash \{0\}) \bigcup (A \oslash B^+)) \bigcap \mathbb{R}^+ \\ &= (A \oslash B^+) \bigcap \mathbb{R}^+ \qquad \text{since } (A \oslash B^-) \bigcap \mathbb{R}^+ = A \oslash_1 \{0\} = \emptyset, \\ &= (A \oslash B^+) \qquad \text{since } A, B^+ \in \mathbb{R}^+, \\ &\in I(\mathbb{R}^+) \subset I(\mathbb{R}) \qquad \text{since } 0 \notin A, B^+. \end{aligned}$$

The proof of the second case proceeds similarly. ∎

An advantage of relational interval arithmetic is that we do not have the division-by-zero problem. For example, the constraint $(\texttt{multiply}^+, \langle(4, +\infty), [0,0], [-3,5)\rangle)$ is reduced to $(\texttt{multiply}^+, \langle(4, +\infty), [0,0], [0,0]\rangle)$, since multiplying any number by zero yields zero.

2.4.5 Disequality

$$\texttt{dif} = \{(x,y) \mid x, y \in \mathbb{R}, x \neq y\}$$

The `dif` relation suffers from the same problem as the `multiply` relation. We partition `dif` into `lt` and `gt`. We have discussed `lt` in section 2.4.2 and interval narrowing for the `gt` relation is similar. In encountering a `dif` constraint, we choose the `lt` partition and perform narrowing. The other partition `gt` is visited upon backtracking or under user control.

2.4.6 Transcendental Functions

$$\texttt{sin} = \{(x,y)\,|\,x,y \in \mathbb{R}, y = \sin(x)\} \qquad \texttt{cos} = \{(x,y)\,|\,x,y \in \mathbb{R}, y = \cos(x)\}$$
$$\texttt{tan} = \{(x,y)\,|\,x,y \in \mathbb{R}, y = \tan(x)\} \qquad \texttt{log} = \{(x,y)\,|\,x,y \in \mathbb{R}, y = \log(x)\}$$

Transcendental functions are important in numerical programming. We discuss a few relations induced by transcendental functions; similar techniques can be applied to others. We can check from figure 2.3 that only the `log` relation satisfies the criterion for interval narrowing, because $\sin^{-1}$, $\cos^{-1}$, and $\tan^{-1}$ are not functions. For example, $\sin^{-1}(0) = k\pi$ for $k = \ldots, -2, -1, 0, 1, 2, \ldots$. We partition the relations `sin`, `cos`, and `tan` in slices of length π in the first attribute:

$$\begin{aligned}
\texttt{sin}_k &= \{(x,y)\,|\,x,y \in \mathbb{R}, x \in I_k, y = \sin(x)\} \\
\texttt{cos}_k &= \{(x,y)\,|\,x,y \in \mathbb{R}, x \in J_k, y = \cos(x)\} \\
\texttt{tan}_k &= \{(x,y)\,|\,x,y \in \mathbb{R}, x \in I_k, y = \tan(x)\}
\end{aligned}$$

where $I_k = \{x\,|\,k\pi + \pi/2 \le x < (k+1)\pi + \pi/2\}$, $J_k = \{x\,|\,k\pi \le x < (k+1)\pi\}$, and $k = \ldots, -2, -1, 0, 1, 2, \ldots$. In addition, we define the interval versions of the transcendental functions. If $I \in I(\mathbb{R})$, then

$$\begin{aligned}
\mathrm{SIN}(I) &= \{\sin(x)\,|\,x \in I\}, & \mathrm{SIN}^{-1}(I) &= \{\sin^{-1}(x)\,|\,x \in I\}, \\
\mathrm{COS}(I) &= \{\cos(x)\,|\,x \in I\}, & \mathrm{COS}^{-1}(I) &= \{\cos^{-1}(x)\,|\,x \in I\}, \\
\mathrm{TAN}(I) &= \{\tan(x)\,|\,x \in I, x \neq k\pi + \pi/2\}, & \mathrm{TAN}^{-1}(I) &= \{\tan^{-1}(x)\,|\,x \in I\}, \\
\mathrm{LOG}(I) &= \{\log(x)\,|\,x \in I\}, & \mathrm{EXP}(I) &= \{\exp(x)\,|\,x \in I\}.
\end{aligned}$$

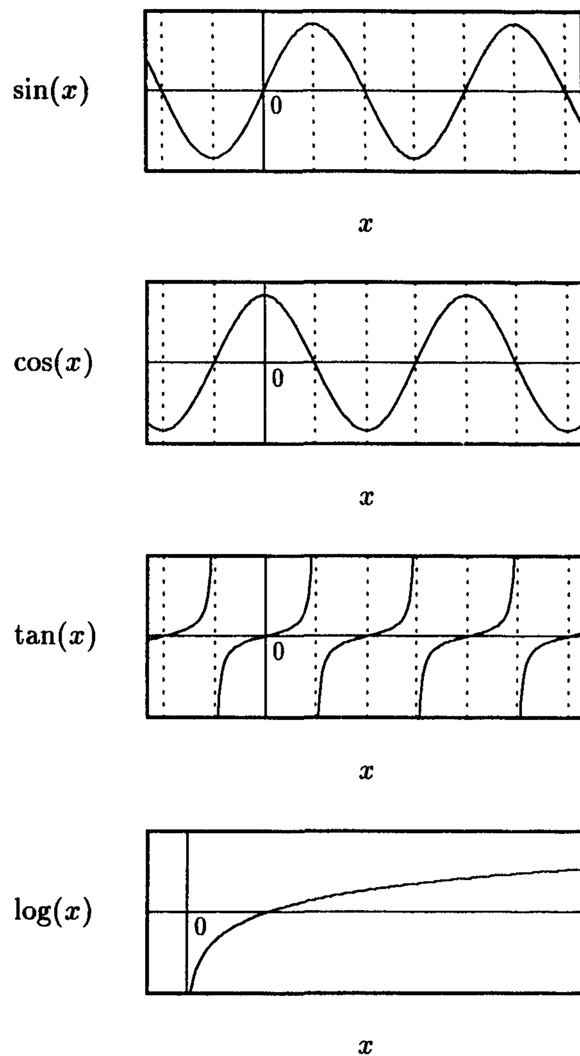

Figure 2.3: Graphs of some transcendental functions.

We are now ready to define the $F_i(p)$'s for $\mathtt{sin}_k$, $\mathtt{cos}_k$, $\mathtt{tan}_k$, and log. Let $I_1, I_2 \in I(\mathbb{R})$. We define

$$\begin{aligned}
F_1(\mathtt{sin}_k)(I_2) &= \mathrm{SIN}^{-1}(I_2) \cap I_k, & F_2(\mathtt{sin}_k)(I_1) &= \mathrm{SIN}(I_1 \cap I_k),\\
F_1(\mathtt{cos}_k)(I_2) &= \mathrm{COS}^{-1}(I_2) \cap J_k, & F_2(\mathtt{cos}_k)(I_1) &= \mathrm{COS}(I_1 \cap J_k),\\
F_1(\mathtt{tan}_k)(I_2) &= \mathrm{TAN}^{-1}(I_2) \cap I_k, & F_2(\mathtt{tan}_k)(I_1) &= \mathrm{TAN}(I_1 \cap I_k),\\
F_1(\mathtt{log})(I_2) &= \mathrm{EXP}(I_2), & F_2(\mathtt{log})(I_1) &= \mathrm{LOG}(I_1).
\end{aligned}$$

It is easy to check that the sine, cosine, and tangent functions and their inverses are continuous in the corresponding partitions. We state without proof the following lemma.

Lemma 2.4.2
If $k = \ldots, -2, -1, 0, 1, 2, \ldots$ and $i = 1, 2$, then the functions $F_i(\mathtt{sin}_k)$, $F_i(\mathtt{cos}_k)$, $F_i(\mathtt{tan}_k)$, and $F_i(\mathtt{log})$ map from real intervals to real intervals. ∎

However, there are infinitely many partitions for sin, cos, and tan. In practice, we restrict the use of sin to the partitions in $-\pi/2$ to $3\pi/2$, cos to those in $-\pi$ to π, and tan to those in $-\pi/2$ to $\pi/2$.

2.5 Constraint Networks

The interval narrowing operation discussed so far applies to individual constraints. In practice, we have more than one constraint in a problem. These constraints may depend on one another by sharing intervals. By naming an interval by a variable and by having a variable occur in more than one constraint, we indicate that constraints share intervals. Note that the material in this section is not related to logic programming but is in conventional notation with destructive assignment. We define an

interval network to be a set of interval constraints. Consider the quadratic equation $v(v-1)=6$. Suppose our initial guess for the positive root of the equation is $[1,100]$. We can express the equation by the following interval network:

$$\{(\texttt{add}, \langle V_1, [1,1], V\rangle), (\texttt{multiply}^{+}, \langle V, V_1, [6,6]\rangle)\},$$

where the variables V and V_1 are initially assigned intervals $[1,100]$ and $(-\infty,+\infty)$ respectively.

Before we present the reduction of interval networks, we discuss the following two observations. First, the reduction of a constraint C in a network affects other constraints that share variables with C. Second, interval narrowing is idempotent as shown in the following lemma.

Lemma 2.5.1
Let $\vec{I} = \langle I_1, \ldots, I_n\rangle$, $\vec{I}' = \langle I'_1, \ldots, I'_n\rangle$, and $\vec{I}'' = \langle I''_1, \ldots, I''_n\rangle$ be tuples of floating-point intervals and p a relation on $\mathbb{R}^n$. If, by interval narrowing, $\vec{I}'$ is obtained from $\vec{I}$ and $\vec{I}''$ is obtained from $\vec{I}'$, then $\vec{I}' = \vec{I}''$.

Proof: To prove the equality of $\vec{I}'$ and $\vec{I}''$, we prove $I'_i = I''_i$ for $i = 1, \ldots, n$. By the definition of interval narrowing and lemma 2.3.2, we have

$$I'_i = \xi(I_i \cap F_i(p)(I_1, \ldots, I_{i-1}, I_{i+1}, \ldots, I_n))$$
$$I''_i = \xi(I'_i \cap F_i(p)(I'_1, \ldots, I'_{i-1}, I'_{i+1}, \ldots, I'_n)).$$

It is obvious that $I''_i \subseteq I'_i$. Next we prove $I'_i \subseteq I''_i$.

Suppose $a'_i \in I'_i$. There exists $a_i \in (I_i \cap F_i(p)(I_1, \ldots, I_{i-1}, I_{i+1}, \ldots, I_n))$ such that $a'_i \in \xi([a_i, a_i])$ by lemma 2.1.5. By the definition of $F_i(p)$, for $j = 1, \ldots, i-1, i+1, \ldots, n$, there exists $a_j \in I_j$ such that $(a_1, \ldots, a_n) \in p$. Since $a_i \in I_i$, we have $a_j \in I'_j$

for each j. Thus, $a_i \in F_i(p)(I'_1, \ldots, I'_{i-1}, I'_{i+1}, \ldots, I'_n))$. This implies that $a_i \in I''_i$. By corollary 2.1.6, $a'_i \in I''_i$. ∎

An interval constraint $(p, \vec{I})$ is *stable* if applying interval narrowing on $\vec{I}$ results in $\vec{I}$. An interval network is *stable* if every constraint in the network is stable. The reduction of an interval network amounts to transforming the network into a stable one. Please note that a stable constraint C in a network may become unstable after the reduction of another constraint that shares intervals with C.

A naive approach for interval network reduction is to reduce each constraint in the network in a round-robin fashion until every constraint becomes stable. As suggested by lemma 2.5.1, this method is inefficient since much computation is wasted in reducing stable constraints. Algorithm 2.1, which is based on the constraint relaxation algorithm described in [15], is the pseudocode of a more efficient procedure. It is similar to the Waltz algorithm [78] and the arc-consistency algorithm AC-3 [46] in that it tries to avoid reducing stable constraints. Without loss of generality, we assume that every constraint in the network is of the form $(p, \langle V_1, \ldots, V_n \rangle)$, where the V_i's are interval-valued variables.

The idea of algorithm 2.1 is to maintain the constraints of the network in two lists: the active list A and the passive list P. The active list contains constraints that are possibly unstable; the passive list contains stable constraints. Initially, the passive list P is empty and the active list A contains all the constraints in the network. Then each constraint in A is reduced using interval narrowing. If any of the intervals are empty after narrowing, the algorithm exits with failure. We say that the network is *inconsistent*. If any interval is narrowed, the algorithm updates the variables. Moreover, constraints in P that share narrowed variables with the reduced constraint may become unstable. The algorithm promotes these constraints from P to A. The

1. initialize A to the list of all constraints in the network
2. initialize P to the empty list
3. **while** A is not empty
4.   remove a constraint $(p, \vec{V})$ from A
5.   apply interval narrowing on $(p, \vec{V})$ to obtain $\vec{V}'$
6.   **if** interval narrowing fails **then**
7.   exit with failure
8.   **else if** $\vec{V} \neq \vec{V}'$ **then**
9.   replace $\vec{V}$ by $\vec{V}'$
10.   **foreach** constraint $(q, \vec{Y})$ in P
11.   **if** $\vec{V}$ and $\vec{Y}$ share narrowed variable(s) **then**
12.   remove $(q, \vec{Y})$ from P and append it to A
13.   **endif**
14.   **endforeach**
15.   **endif**
16.   append $(p, \vec{V})$ to the end of P
17. **endwhile**

Algorithm 2.1: A Relaxation Algorithm.

reduced constraint is stable and is appended to the end of P. This process repeats until A is empty.

Algorithm 2.1 resembles a classical iterative numerical-approximation technique called "relaxation" [67], which was adopted in a constraint system in [69]. In relaxation, we make an initial guess at the values of the unknowns and then estimate the error of the guess. New guesses are then made using the original guess and the error estimate. This process is repeated until the error is sufficiently small or satisfies a certain termination criterion. In algorithm 2.1, the initial guess is the input intervals. Errors are values in the intervals that cannot satisfy the relation associated with the constraint. New intervals are computed by removing the inconsistent values. These narrowed intervals are better approximations of the originals in the sense that they

provide more accurate inclusions of the answers. Numerical relaxation may fail to converge or terminate even when the constraints have a solution. Algorithm 2.1 does not suffer from this problem as shown by the following theorem.

Theorem 2.5.2

Algorithm 2.1 always terminates. The resulting interval network is either inconsistent or stable.

Proof: Algorithm 2.1 halts either when interval narrowing of a constraint fails or the list A becomes empty. In the former case, the network is inconsistent. For the latter case, we observe that the size of list A decreases after each iteration of the algorithm unless variables are narrowed and constraints are moved from P to A. However, the precision of a floating-point system is finite and thus interval narrowing cannot occur indefinitely due to the use of outward-rounding. Therefore, list A must become empty after a finite number of iterations. All constraints of the network are now in P and are stable. Thus the network is stable. ∎

In the following, we show how algorithm 2.1 finds the positive root of the equation $v(v-1)-6=0$ with initial guess $v \in [1,100]$. Initially, the passive list of constraints, P, is empty and the active list, A, contains both constraints

$$A = [C_1, C_2] = [(\mathbf{add}, \langle V_1, [1,1], V\rangle), (\mathtt{multiply}^+, \langle V, V_1, [6,6]\rangle)],$$

where $V = [1,100]$ and $V_1 = (-\infty, +\infty)$. We remove $C_1 = (\mathbf{add}, \langle V_1, [1,1], V\rangle)$ from A and reduce it by interval narrowing as shown in table 2.1. The updated values of V and V_1 are $[1,100]$ and $[0,99]$ respectively. Since C_1 has become stable, we append it to P. Thus A becomes $[C_2]$ and P becomes $[C_1]$. Next we remove C_2 from A and reduce it as shown in table 2.2. The values of V and V_1 become $[1,100]$ and $[0.06,6]$ respectively. Since V_1 is narrowed and C_1 shares V_1 with C_2, we promote C_1 from P

Input Intervals = $\langle I_1, I_2, I_3 \rangle$	$(-\infty, +\infty)$	$[1, 1]$	$[1, 100]$
$\xi(F_1(\texttt{add})) = \xi([1, 100] \ominus [1, 1])$	$[0, 99]$		
$\xi(F_2(\texttt{add})) = \xi([1, 100] \ominus (-\infty, +\infty))$		$(-\infty, +\infty)$	
$\xi(F_3(\texttt{add})) = \xi((-\infty, +\infty) \oplus [1, 1])$			$(-\infty, +\infty)$
Output Intervals = $\langle I'_1, I'_2, I'_3 \rangle$	$[0, 99]$	$[1, 1]$	$[1, 100]$

Table 2.1: Interval narrowing of $(\texttt{add}, \langle V_1, [1, 1], V \rangle)$.

Input Intervals = $\langle I_1, I_2, I_3 \rangle$	$[1, 100]$	$[0, 99]$	$[6, 6]$
$\xi(F_1(\texttt{multiply}^+)) = \xi([1, 100] \otimes [0, 99])$			$[0, 9900]$
$\xi(F_2(\texttt{multiply}^+)) = \xi([6, 6] \oslash_2 [1, 100])$		$[0.06, 6]$	
$\xi(F_3(\texttt{multiply}^+)) = \xi([6, 6] \oslash_1 [0, 99])$	$(0.0606, +\infty)$		
Output Intervals = $\langle I'_1, I'_2, I'_3 \rangle$	$[1, 100]$	$[0.06, 6]$	$[6, 6]$

Table 2.2: Interval narrowing of $(\texttt{multiply}^+, \langle V, V_1, [6, 6] \rangle)$.

to A. C_2, which has become stable, is appended to A. This process repeats until the precision of the underlying floating-point system is reached and no more narrowing takes place. In section 4.3, we describe a prototype that executes algorithm 2.1. In this example, our prototype takes 42 steps to reach the stable state. The history of the values of A, P, V, and V_1, with four significant digits, after each narrowing step is summarized in table 2.3.

If we try the same example with initial guess $V \in [50, 100]$, algorithm 2.1 produces the trace in table 2.4. The network is inconsistent. Failure is useful in constraint logic programming. A search tree has internal nodes as choice points and external nodes as failure or success points. By discovering inconsistency in the constraint network,

A	P	V	V_1
$[C_1, C_2]$	$[]$	$[1, 100]$	$(-\infty, +\infty)$
$[C_2]$	$[C_1]$	$[1, 100]$	$[0, 99]$
$[C_1]$	$[C_2]$	$[1, 100]$	$[0.06, 6]$
$[C_2]$	$[C_1]$	$[1.06, 7]$	$[0.06, 6]$
$[C_1]$	$[C_2]$	$[1.06, 7]$	$(0.8571, 5.661)$
$[C_2]$	$[C_1]$	$(1.857, 6.661)$	$(0.8571, 5.661)$
$[C_1]$	$[C_2]$	$(1.857, 6.661)$	$(0.9009, 3.231)$
$[C_2]$	$[C_1]$	$(1.900, 4.231)$	$(0.9009, 3.231)$
$[C_1]$	$[C_2]$	$(1.900, 4.231)$	$(1.418, 3.157)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$[]$	$[C_1, C_2]$	$(2.999, 3.001)$	$(1.999, 2.001)$

Table 2.3: Traces of A, P, V, and V_1.

we turn choice points (internal nodes) into failure points (external nodes) and prune the search tree.

2.6 Domain Splitting

Algorithm 2.1 is "incomplete" [46]: a network can be stable without either a solution or inconsistency being found. In the finite domain case, enumeration, instantiation, and backtracking can be used to find a particular solution after the constraint network becomes stable. This method is infeasible for interval domains, which are infinite sets. We use domain splitting [74] in place of enumeration and instantiation. When an interval network becomes stable, we split an interval into two partitions, choose

A	P	V	V_1
$[C_1, C_2]$	$[]$	$[50, 100]$	$(-\infty, +\infty)$
$[C_2]$	$[C_1]$	$[50, 100]$	$[49, 99]$
$[C_1]$	$[C_2]$	$\emptyset$	$\emptyset$

Table 2.4: Reduction of an inconsistent network.

one partition and visit the other upon automatic backtracking or user control.

Chapter 3

Extending CHIP With Interval Arithmetic

So far, we have explained how a *static* network of interval constraints in terms of floating-point intervals can be made stable using algorithm 2.1. We have not considered how such networks can be specified. The number of constraints and variables in a network can be large. This makes the specification a tedious and error-prone task. We look for a programming language that can (1) describe interval constraints, queries, intervals, answers, and variables, in a coherent manner, and (2) describe an interval network concisely. CHIP [22], based on Horn clauses with domain variables [74] and consistency techniques [46][1], satisfies those requirements. Both domains (page 93) and intervals are sets in which a solution, if one exists, must lie. ***This suggests that we view intervals as domains and view interval constraints as domain constraints (page 95) in CHIP.*** An interval network can be represented implicitly in a CHIP program and the parts of the network are generated dynamically during derivation.

[1] **Please refer to appendix A for a brief introduction to the concepts and terminologies of consistency techniques in logic programming.**

Domains in CHIP are finite sets of constants. The restriction to finite sets allows a simple implementation based on enumeration. We extend domains to infinite sets that should satisfy the following criteria. First, the infinite sets can be represented by a finite and small amount of resources. Second, there is a terminating algorithm to compute the intersection of these infinite sets, since intersection is an essential operation in the unification of domain variables (page 94). Third, there is a terminating algorithm to implement the Looking-Ahead Inference Rule (LAIR) [74] (page 96) for reducing domain constraints with infinite domains. It is easy to verify that intervals satisfy the first two criteria. For the third criterion, we show in the following sections that the interval narrowing operation is an implementation of LAIR for interval domains.

3.1 Approximating An Arithmetic Relation

In logic programming, it is essential to know the relations denoted by the predicates in a program. In Prolog and CLP($\mathcal{R}$) [36], the meaning of the arithmetic predicates is defined in terms of real arithmetic. The floating-point implementations of these predicates, however, are incorrect with respect to the definitions of the predicates. We cannot assume real arithmetic on real computers. The effect of rounding or truncation should be captured in the definition of arithmetic predicates.

Let $a < b$ be two adjacent floating-point numbers. Thus $\xi([x, x]) = (a, b)$ for all $x \in (a, b)$. In other words, numbers in (a, b) are indistinguishable under the floating-point system $\mathbb{F}$. We define the *approximation* p^ϵ of a relation $p \subseteq \mathbb{R}^n$ by

$$p^\epsilon = \{(x_1, \ldots, x_n) \mid \exists (x'_1, \ldots, x'_n) \in p \text{ s.t. } x_i \in \xi([x'_i, x'_i]) \text{ for } i = 1, \ldots, n\}.$$

Intuitively, p^ϵ is the union of the neighborhood of all points in p.

3.2 Interval Narrowing As LAIR

We show that interval narrowing and the Looking-Ahead Inference Rule perform the same amount of narrowing on the input intervals.

Let $X_1^{I_1}, \ldots, X_n^{I_n}$ be domain variables with interval domains $I_1, \ldots, I_n$. Reducing the domain constraint $p^\epsilon(X_1^{I_1}, \ldots, X_n^{I_n})$ with LAIR [74] (page 96) yields new domains $E_1, \ldots, E_n$, where

$$E_i = \{x_i \in I_i \mid \exists\, x_j \in I_j (j = 1, \ldots, n; j \neq i) \text{ s.t. } (x_1, \ldots, x_n) \in p^\epsilon\}.$$

Theorem 3.2.1

Let $I_1, \ldots, I_n$ be floating-point intervals and $p \in \mathbb{R}^n$. If $I'_1, \ldots, I'_n$ are output intervals of performing interval narrowing on $(p, \langle I_1, \ldots, I_n \rangle)$ and $E_1, \ldots, E_n$ are new domains obtained by applying LAIR to $p^\epsilon(X_1^{I_1}, \ldots, X_n^{I_n})$, then $I'_i = E_i$ for all $i = 1, \ldots, n$.

Proof: Let $J_i = \{x_i \mid (x_1, \ldots, x_n) \in ((I_1 \times \cdots \times I_n) \bigcap p)\}$.

$$\begin{aligned}
I'_i &= \xi(J_i) \qquad \text{by theorem 2.3.3 (page 27)} \\
&= \{x_i \mid \exists x'_i \in J_i \text{ s.t. } x_i \in \xi([x'_i, x'_i])\} \qquad \text{by lemma 2.1.5 (page 21)} \\
&= \{x_i \mid (x'_1, \ldots, x'_n) \in ((I_1 \times \cdots \times I_n) \bigcap p) \text{ s.t. } x_i \in \xi([x'_i, x'_i])\} \\
&= \{x_i \in I_i \mid \exists\, x_j \in I_j, j = 1, \ldots, n, j \neq i \text{ s.t. } (x_1, \ldots, x_n) \in p^\epsilon\} \\
&\qquad\qquad \text{since } \xi([x'_j, x'_j]) \subseteq I_j \\
&= E_i.
\end{aligned}$$

■

There are two implications of theorem 3.2.1.

First, interval narrowing is an instance of LAIR for interval domains. Thus interval arithmetic based on interval narrowing can be incorporated

in CHIP without changing its operational semantics. Second, interval narrowing is the basis of an implementation of LAIR for interval domains.

Programmers using the extended language (page 50), however, should be aware that the arithmetic predicates are defined in terms of the approximations p^{ϵ} of the corresponding relations $p \subseteq \mathbb{R}^n$.

3.3 Constraint Relaxation

The other essential component of relational interval arithmetic is algorithm 2.1. We show that algorithm 2.1 (page 39) is already embedded in the proof procedure of CHIP [74].

CHIP uses three extra inference rules on top of SLDD-resolution (page 95): the *Forward Checking Inference Rule* (FCIR), the *Looking-Ahead Inference Rule* (LAIR), and the *Partial Looking-Ahead Inference Rule* (PLAIR). FCIR is a special case of LAIR and PLAIR is an approximation of LAIR. In the following, we restrict our discussion to the proof procedure involving LAIR only.

Derivations in CHIP consist of SLDD-derivation steps interleaved with LAIR. In logic programming, the computation rule is usually crucial. Prolog always selects the leftmost atom in a goal. This same computation rule is not suitable for CHIP because the leftmost goal may be a stable constraint. As interval narrowing and, thus, LAIR are idempotent, such selection would produce no effective computation. CHIP uses a computation rule that tries to avoid selecting stable constraints.

CHIP maintains a goal by two sets Y and N of distinct atoms. The set N contains all the atoms submitted to a lookahead declaration (page 97) that, if selected, will not change under the application of LAIR. Y is the set of all the other atoms. Initially,

Y is the set of all atoms in the original goal and N is the empty set. We denote a goal by $\leftarrow \langle Y, N \rangle$; Y is called the *Y-part* of the goal and N is the *N-part*. CHIP uses the lookahead-efficient computation rule in its proof procedure.

A computation rule R is *lookahead-efficient* if and only if:

1. an atom submitted to a lookahead declaration is selected by R only when it is ground or lookahead-available (page 97);
2. R selects only atoms in the Y-part of the goal; and
3. if one or more atoms submitted to a lookahead declaration in the Y-part are lookahead-available or ground, then R selects one of them.

Intuitively, an atom A is lookahead-available if it is in a suitable form to be reduced by LAIR. It requires that A is lookahead-checkable (page 96). That is because LAIR needs to check the satisfiability of ground instances of A using some form of resolution. There is no such need in the case of interval narrowing and we can drop the requirement. For $p^{\epsilon} \subseteq \mathbb{R}^n$, we declare

lookahead $p^{\epsilon}(d, \ldots, d)$.

Therefore, the atom $p^{\epsilon}(X_1^{I_1}, \ldots, X_n^{I_n})$ is always lookahead-available.

The proof procedure of CHIP [74] is as follows. Let $\leftarrow \langle Y, N \rangle$ be a goal and G be the atom selected by a lookahead-efficient computation rule. Let θ be the substitution resulting from the reduction of G by either a SLDD-derivation step or LAIR and I the set of atoms introduced by the reduction of G. I is the empty set if LAIR is used and the body of the input clause otherwise. Let YI be $Y \cup I \setminus \{G\}$ and NY the set of atoms Q in N such that $Q\theta \neq Q$. The new goal is

1. $\leftarrow \langle YI \cup NY, \{G\} \cup N \setminus NY \rangle \theta$ if G is lookahead-available and $G\theta$ contains more than one domain variable;

2. $\leftarrow \langle YI \cup NY, N \setminus NY \rangle \theta$ otherwise;

where $\langle X, Y \rangle \theta$ is the application of θ to all the atoms of X and Y.

Theorem 3.3.1

The proof procedure of CHIP, using a lookahead-efficient computation rule, executes algorithm 2.1.

Proof: We show the truth of the theorem by the following arguments:

1. In algorithm 2.1, the active list A is contained in the Y-part of the goal and the passive list P is the N-part.

2. LAIR, as proved in theorem 3.2.1, performs interval narrowing (line 5 of algorithm 2.1).

3. NY is the set of atoms that become potentially unstable since they share variables, the domains of which are narrowed, with the current reduced atom. I is the empty set since LAIR is used. The computation of the new Y-part $YI \cup NY = Y \setminus \{G\} \cup NY$ is equivalent to line 4 and lines 10 to 14 of algorithm 2.1. The computation of the new N-part is equivalent to in lines 10 to 14 and line 16.

4. A lookahead-efficient computation rule ensures that if there exist constraints in the Y-part, then they will be selected. This executes the outer loop (lines 3 to 17) of algorithm 2.1 completely before other atoms are selected.

∎

3.4 The ICHIP Language

ICHIP is an extension of CHIP with relational interval arithmetic. Domains in ICHIP can also be intervals. Reduction of goals with arithmetic primitive predicates are performed by interval narrowing. The proof procedure of ICHIP consists of SLDD-derivation steps interleaved with complete executions of algorithm 2.1.

We express the interval constraint $(p, \langle I_1, \ldots, I_n \rangle)$ in ICHIP as

$$p^{\epsilon}(X_1^{I_1}, \ldots, X_n^{I_n}).$$

Constraints share intervals when they share domain variables. For example, to solve the positive root of $x(x-1) - 6 = 0$ with initial interval $[1, 100]$, we pose the query:

$$\leftarrow \mathtt{add}^{\epsilon}(X_1^{(-\infty,+\infty)}, 1, X^{[1,100]}), \mathtt{multiply}^{+\epsilon}(X^{[1,100]}, X_1^{(-\infty,+\infty)}, 6).$$

Partitioning of relations can also be expressed compactly in ICHIP. For example, the `multiply` relation can be defined by

$$\mathtt{multiply}^{\epsilon}(X, Y, Z) \leftarrow \mathtt{le}^{\epsilon}(0, X), \mathtt{multiply}^{+\epsilon}(X, Y, Z).$$
$$\mathtt{multiply}^{\epsilon}(X, Y, Z) \leftarrow \mathtt{lt}^{\epsilon}(X, 0), \mathtt{multiply}^{-\epsilon}(X, Y, Z).$$

3.5 Domain Splitting And Answer Interpretation

It is well-known that LAIR, based on the arc consistency technique, is "incomplete" [46]: an interval network can be stable but neither a solution nor inconsistency is found. Therefore, an ICHIP derivation can end with *floundered goals* [49], goals with atoms that cannot be selected by the computation rule. For example, this behavior may occur if we use ICHIP to solve the roots of general polynomial and

simultaneous equations [15]. In CHIP, floundering is avoided by the built-in predicate **`indomain`**, which implements a form of case analysis. The **`indomain`** predicate takes as argument a domain variable, and enumerates the elements in the domain of its argument. Intervals are infinite sets. Implementing a similar predicate is not feasible. For interval domains, we use another case analysis technique: ***domain splitting***. Cleary [15] discusses two predicates, `linear_split` and **`exp_split`**, that split an interval into two partitions, visit one of them, and visit the other upon backtracking. `Linear_split` and `exp_split` split at different points of the interval. CHIP also provides a similar built-in predicate `split`, which handles finite domains. This can easily be extended to handle interval domains. We can use domain splitting to narrow the interval domains until they reach an acceptable width or the smallest width allowed by the precision of the underlying floating-point system.

Domain splitting can be used to narrow intervals to a desirable width but it may not find solutions either. Floundering can still occur. We call the floundered goals *incomplete solutions*. Logically, incomplete solutions can be interpreted as qualified or conditional answers [75, 14]. Let P be a logic program with domain variables and $\leftarrow G$ a goal. Suppose, we have derived from $\leftarrow G$ a non-empty goal $\leftarrow G_1, \ldots, G_n$, with θ being the composition of all substitutions so far. The clause $(G \leftarrow G_1, \ldots, G_n)\theta$ is a ***conditional answer*** to the original goal. From the soundness of SLDD-resolution and LAIR [74], we have

$$P \models \forall(G \leftarrow G_1, \ldots, G_n)\theta$$

where the universal quantification is over all variables in $(G \leftarrow G_1, \ldots, G_n)\theta$. This interpretation is also adopted for answer constraints in CLP languages [33].

Chapter 4

Extending CLP($\mathcal{R}$) With Interval Arithmetic

CLP($\mathcal{R}$)[1] is another suitable language for specifying interval networks. An interval can be represented by a pair of inequalities. For example,

$$1.5 < X \wedge X \leq 3.7$$

says that X belongs to $(1.5, 3.7]$. *This suggests that we view intervals as inequality constraints in CLP($\mathcal{R}$).*

Example 4.0.1

We can represent the interval constraint $(\texttt{multiply}^{+}, \langle [0,2], [1,2.5], [4,6] \rangle)$ in CLP($\mathcal{R}$) by

$$0 \leq X, X \leq 2, 1 \leq Y, Y \leq 2.5, 4 \leq Z, Z \leq 6, X * Y = Z.$$

[1] Please refer to appendix B for a brief introduction to the CLP scheme.

However, if we submit the above example as a query to CLP($\mathcal{R}$) Version 1.1 [28], we get an answer constraint that is exactly the same as the query. This is because CLP($\mathcal{R}$) delays the evaluation of nonlinear constraints. A more useful answer constraint is

$$1.6 \leq X, X \leq 2, 2 \leq Y, Y \leq 2.5, 4 \leq Z, Z \leq 5, X * Y = Z.$$

■

Example 4.0.2

The interval constraint $(\mathbf{add}, \langle [0,2], [1,3], [4,6] \rangle)$ is represented by

$$0 \leq X, X \leq 2, 1 \leq Y, Y \leq 3, 4 \leq Z, Z \leq 6, X + Y = Z.$$

CLP($\mathcal{R}$) [28] returns

$$X = Z - Y, Y \leq Z, Z \leq 2 + Y, Y \leq 3, 1 \leq Y, Z \leq 6, 4 \leq Z,$$

which has some resemblance to the original query. Some arithmetic manipulation confirms that the answer constraint is equivalent to the original one. A more useful answer constraint is

$$1 \leq X, X \leq 2, 2 \leq Y, Y \leq 3, 4 \leq Z, Z \leq 5, X + Y = Z.$$

■

The examples suggest that CLP($\mathcal{R}$), although adequate to express interval constraints syntactically, fails to narrow intervals. We enhance CLP($\mathcal{R}$) with interval narrowing (page 27) and algorithm 2.1 (page 39). To establish the theoretical basis of the enhancement, we modify the basic CLP scheme by inserting a constraint simplification step and show that interval narrowing is a constraint simplification operation.

4.1 The Modified CLP Scheme

The declarative semantics remains unchanged in the extended scheme. The modified operational semantics is a generalization of $\mathcal{M}_\mathcal{X}$ derivations [32] (page 103). Let P be a CLP($\mathcal{X}$) program (page 103), where $\mathcal{X}$ is a domain of computation with structure $\mathcal{M}_\mathcal{X}$ (page 99), and $\leftarrow G_i$ be a goal. $\leftarrow G_{i+1}$ is ***derived in $\mathcal{M}_\mathcal{X}$ in a generalized sense*** from $\leftarrow G_i$ if

1. $\leftarrow G'$ is derived in $\mathcal{M}_\mathcal{X}$ from $\leftarrow G_i$, and
2. $\leftarrow G_{i+1} = \nu(\leftarrow G')$, where ν is a *normal-form function* that maps from goal to goal such that $P \models_{\mathcal{M}_\mathcal{X}} \exists(G') \Leftrightarrow P \models_{\mathcal{M}_\mathcal{X}} \exists(G_{i+1})$.

A *generalized $\mathcal{M}_\mathcal{X}$ derivation* is a sequence of goals $G = G_0, G_1, G_2, \ldots$, possibly infinite, such that G_{i+1} is derived in $\mathcal{M}_\mathcal{X}$ in a generalized sense from G_i. A generalized $\mathcal{M}_\mathcal{X}$ derivation is *successful* if it is finite and the last goal contains no atoms. The soundness and completeness of generalized $\mathcal{M}_\mathcal{X}$ derivations follow directly from the soundness and completeness of $\mathcal{M}_\mathcal{X}$ derivations and the definition of the normal-form function. A generalized $\mathcal{M}_\mathcal{X}$ derivation is *finitely-failed* if it is finite, the last goal has one or more atoms, and condition 1 does not hold.

The generalized $\mathcal{M}_\mathcal{X}$ derivation step is not new. In fact, it has been implemented in CLP($\mathcal{R}$) for constraint simplification and output [34]. The $\mathcal{M}_\mathcal{X}$ derivation step only checks the solvability of the constraint accumulated so far. Therefore, the answer constraint of a successful $\mathcal{M}_\mathcal{X}$ derivation is usually complex and difficult to interpret. A useful system should simplify the constraint to a more readable form. For example, CLP($\mathcal{R}$) simplifies the constraint $\{X + Y = 4, X - Y = 1\}$ to $\{X = 2.5, Y = 1.5\}$. Suppose the goal $\leftarrow c', \vec{A}'$ is derived in $\mathcal{M}_\mathcal{X}$ from $\leftarrow c, \vec{A}$. CLP($\mathcal{R}$) simplifies c' to c''

such that

$$\models_{\mathcal{M}_{\mathcal{X}}} \exists(c') \Leftrightarrow \models_{\mathcal{M}_{\mathcal{X}}} \exists(c'')$$

and thus

$$P \models_{\mathcal{M}_{\mathcal{X}}} \exists(c', \vec{A}') \Leftrightarrow P \models_{\mathcal{M}_{\mathcal{X}}} \exists(c'', \vec{A}');$$

CLP($\mathcal{R}$) is based on the generalized $\mathcal{M}_{\mathcal{X}}$ derivation.

Theorem 4.1.1

If $\mathcal{C}'$ is obtained from $\mathcal{C}$ using interval narrowing on p, where $\mathcal{C}$ is $X_1 \in I_1, \ldots, X_n \in I_n, p(X_1, \ldots, X_n)$ and $\mathcal{C}'$ is $X_1 \in I_1', \ldots, X_n \in I_n', p(X_1, \ldots, X_n)$, then

$$\models_{\mathcal{M}_{\mathcal{X}}} \exists(\mathcal{C}) \Leftrightarrow \models_{\mathcal{M}_{\mathcal{X}}} \exists(\mathcal{C}').$$

Proof: From theorem 2.3.1 (page 26). ∎

Theorem 4.1.1 guarantees that interval narrowing transforms an interval constraint into a stable one with the same solution space. Algorithm 2.1, which performs narrowing repeatedly on interval constraints in an interval network, is therefore a normal-form function.

4.2 ICLP($\mathcal{R}$)

ICLP($\mathcal{R}$) is an extension of CLP($\mathcal{R}$) with relational interval arithmetic. An interval constraint $(p, \langle I_1, \ldots, I_n \rangle)$ is expressed in ICLP($\mathcal{R}$) as

$$X_1 \in I_1, \ldots, X_n \in I_n, p(X_1, \ldots, X_n),$$

where $X_i \in I_i$ is an appropriate pair of inequalities. Constraints share intervals when they share logical variables. ICLP($\mathcal{R}$) and CLP($\mathcal{R}$) have the same syntax

and declarative semantics. Operationally, ICLP($\mathcal{R}$) is based on the generalized $\mathcal{M}_{\mathcal{X}}$ derivation, using algorithm 2.1 as the normal-form function. For example, to solve the positive root of $x(x-1)-6=0$ with initial interval [1, 100], we pose the query:

$$\leftarrow X \geq 1, X \leq 100, \mathtt{add}(X_1, 1, X), \mathtt{multiply}(X, X_1, 6).$$

Again, partitioning of relations can be expressed compactly in ICLP($\mathcal{R}$). For example, the **multiply** relation can be defined by

$$\mathtt{multiply}(X, Y, Z) \leftarrow X \geq 0, \mathtt{multiply}^{+}(X, Y, Z).$$
$$\mathtt{multiply}(X, Y, Z) \leftarrow X < 0, \mathtt{multiply}^{-}(X, Y, Z).$$

We adopt the same domain splitting strategy and answer interpretation as described in section 3.5 (page 50).

4.3 An ICLP($\mathcal{R}$) Interpreter

We describe an ICLP($\mathcal{R}$) interpreter written in CLP($\mathcal{R}$). There are two advantages to use CLP($\mathcal{R}$) as the implementation language. First, ordinary arithmetic constraint solving can be handled by CLP($\mathcal{R}$). We do not need to re-invent the CLP($\mathcal{R}$) constraint solver. Second, CLP($\mathcal{R}$) can keep track of the interval bounds automatically. The interpreter can use the meta predicate "**coded_ccs**" [29], implemented as the arity 3 "**dump**" predicate in [28], to examine interval bounds.

We start by presenting a CLP($\mathcal{R}$) meta-interpreter and show how it can be modified to an ICLP($\mathcal{R}$) interpreter. Just as we can hide unification in a Prolog meta-interpreter, we can also hide constraint solving in a CLP($\mathcal{R}$) meta-interpreter. In fact, our CLP($\mathcal{R}$) meta-interpreter, shown in program 4.1, is the same as a Prolog meta-interpreter except that CLP($\mathcal{R}$) rules are encoded differently. Each CLP($\mathcal{R}$)

```
solve([]).
solve([Goal|Goals]) ←
    solveGoal(Goal),
    solve(Goals).

solveGoal(Goal) ←
    clause(Goal,Body),
    solve(Body).
```

Program 4.1: A CLP($\mathcal{R}$) meta-interpreter.

rule

```
Head ← Constraints, Body.
```

is stored as a clause of the form

```
clause(Head, Body) ← Constraints.
```

where **Head** is an atom, **Constraints** is a list of constraints, and **Body** is a list of atoms. For example, a program to compute Fibonacci numbers is written as:

```
clause(fib(0,1),[]).
clause(fib(1,1),[]).
clause(fib(N,X1+X2),[fib(N-1.X1),fib(N-2,X2)]) ← N > 1.
```

We use the query ← `solve([fib(14,X)])` to compute the fourteenth Fibonacci number.

The ICLP($\mathcal{R}$) interpreter distinguishes between arithmetic constraints that are intended for interval narrowing and those that are not. In this way the user has the choice of using interval narrowing or not. For example, the constraints `X+Y=Z` and `add(X,Y,Z)` have the same declarative meaning but differ operationally. We apply interval narrowing on the latter only; the CLP($\mathcal{R}$) constraint solver handles

the former. An ICLP($\mathcal{R}$) rule is of the form

$$\texttt{Head} \leftarrow \texttt{Constraints}, \texttt{IConstraints}, \texttt{Body}.$$

and is stored as

$$\texttt{clause(Head}, \texttt{IConstraints}, \texttt{Body)} \leftarrow \texttt{Constraints}.$$

where **Head** is an atom, **Body** is a list of atoms, **Constraints** is a list of ordinary CLP($\mathcal{R}$) constraints, and **IConstraints** is a list of arithmetic constraints intended for interval narrowing. We extend the CLP($\mathcal{R}$) meta-interpreter in program 4.1 with the relaxation algorithm. The top level of the modified interpreter is shown in program 4.2. The **dumpIntConstr** predicate transforms the interval constraints into their equivalent forms in CLP($\mathcal{R}$), such as add(X, Y, Z) to $X + Y = Z$. It is invoked at the end of a derivation. Algorithm 2.1 is encoded in the **reduceNet** predicate, which calls the **dump** predicate and the narrowing routines for the arithmetic primitives. Our current ICLP($\mathcal{R}$) prototype is incomplete since outward-rounding has not been included. To complete the implementation, we have to provide additional arithmetic primitives in CLP($\mathcal{R}$) that control the rounding mode of the underlying floating-arithmetic hardware.

To find the positive root of $x^2 - x - 6 = 0$ in our prototype, we pose the query:

```
← X >= 0,X <= 100,solve([add(X1,1,X),multiply(X,X1,6)],[]).
```

Table 2.3 is derived from a trace in response to this query, except that the outward rounding has been added manually.

```
% solve(INet,Goals) is true if the goal ← INet,Goals has a
%   successful generalized M_ℜ derivation, where INet is an
%   interval network and Goals is a list of atoms.
solve(INet,Goals) ←
    reduceNet(INet,[],INet1),
    solve1(Goals,INet1,INet2),
    dumpIntConstrs(INet2).

% solve1(Goals,INet1,INet2) is true if the goal ← INet1,Goals
%   has a successful generalized M_ℜ derivation and INet2 is a
%   stable interval network resulting from the derivation, where
%   INet1 is an interval network and Goals is a list of atoms.
solve1([],INet,INet).
solve1([Goal|Goals],Inet,INet2) ←
    solveGoal(Goal,Inet,INet1),
    solve1(Goals,INet1,INet2).

% solveGoal(Goal,PList,PList2) is true if there is a generalized
%   M_ℜ derivation from ← PList,Goal to ← PList2.  Goal is an
%   atom, and PList and PList2 are stable interval networks.
solveGoal(Goal,PList,PList2) ←
    clause(Goal,AList,Body),
    reduceNet(AList,PList,PList1),
    solve1(Body,PList1,PList2).

% reduceNet(AList,PList,PList2) is true if PList2 is the stable
%   version of the interval network with active list AList and
%   passive list PList.
reduceNet([],PList,PList).
reduceNet([IConstr|AList],PList,PList1) ←
    reduceNet(IConstr,SqVars),
    reduceNet1(SqVars,IConstr,AList,PList,PList1).
```

Program 4.2: The top level of an ICLP($\mathcal{R}$) interpreter in CLP($\mathcal{R}$).

Chapter 5

Analysis Of The Domain Restriction Operations

The interval narrowing operation plays an important role in relational interval arithmetic. Theorem 3.2.1 (page 46) shows that interval narrowing is equivalent to LAIR, which is a domain restriction operation. We present a general framework for the domain restriction operations, based on a generalization of cylindrical closure and cylindrance [5]. We show that the algorithms of some of our predecessors are instances of our scheme. Before we proceed, we clarify our choice of notation and results in basic set theory, adopted from [24].

5.1 Basic Set Theory

If X and Y are sets, a *function* f from X to Y is a set of ordered pairs such that for each $x \in X$ there is a unique element $y \in Y$ with $(x, y) \in f$. We write $y = f(x)$. X is called the *domain* of f and Y is the *codomain* of f. The *range* of f consists of the

elements $y \in Y$ for which there exists an $x \in X$ such that $f(x) = y$. We denote by $X \to Y$ the set of all functions from X to Y. Let $f \in (Y \to Z)$ and $X \subseteq Y$. We define the *restriction* of f to X, denoted by $f \downarrow X$, by

$$(f \downarrow X)(x) = f(x) \text{ for all } x \in X.$$

In the rest of the chapter, we use a special kind of functions, the ranges of which are more important than the functions themselves. These functions are so distinguishable that they deserve new terminology and notation. Suppose that $x \in (I \to X)$ is such a function. We call x a *family*. I is the *index set* and $i \in I$ is an index. We denote the value of x at an index i by x_i, which is a *term* of the family. In this case, we speak of a family x in X.

We use the following notation in this chapter. I and U are non-empty sets. Let $E \in (I \to \mathcal{P}(U))$ be a family in subsets of U, where $\mathcal{P}(U)$ is the power set of U. Each E_i is the set of all possible values for *attribute* $i \in I$. Let $J \subseteq I$. We define the *generalized Cartesian product* of S by

$$\bigotimes E = \{e \mid e \in (J \to U) \wedge \forall j \in J : e_j \in E_j\}.$$

Each element of $\bigotimes(E \downarrow J)$ is a *J-tuple*, or simply *tuple*, with attributes in J. A *relation p with arity J* is a set of J-tuples. Thus $p \subseteq \bigotimes(E \downarrow J)$. Our notion of arity is more general than the conventional definition, which only specifies the number of arguments of a relation; our definition also specifies the indices of the arguments.

Let J and K be subsets of I. If p is a relation with arity J and q is a relation with arity K, then the *natural join* of p and q is a relation with arity $J \cup K$ defined by

$$p \bowtie q = \{z \in \bigotimes(E \downarrow (J \cup K)) \mid \exists x \in p, y \in q \text{ s.t. } z_l = x_l \text{ if } l \in J \wedge z_l = y_l \text{ if } l \in K\}.$$

When $l \in J \cap K$, the definition enforces that $x_l = z_l = y_l$. The following lemma is an equivalent definition of natural join.

Lemma 5.1.1
$z \in (p \bowtie q) \Leftrightarrow (z \in \bigotimes(E \downarrow (J \cup K)) \wedge (z \downarrow J) \in p \wedge (z \downarrow K) \in q)$.

Proof:

$$\begin{aligned} & z \in p \bowtie q \\ \Leftrightarrow \quad & z \in \bigotimes(E \downarrow (J \cup K)) \wedge \exists x \in p, y \in q \text{ s.t. } (z_j = x_j \text{ if } j \in J \wedge z_j = y_j \text{ if } j \in K) \\ \Leftrightarrow \quad & z \in \bigotimes(E \downarrow (J \cup K)) \wedge (z \downarrow J) \in p \wedge (z \downarrow K) \in q. \end{aligned}$$

■

If p is a family in relations, then $\bowtie p$ denotes the natural join of all terms (page 61) of p.

We are now ready to define two important operations. If $J \subseteq K \subseteq I$ and p is a relation with arity K, then the *projection of p on J* is defined as

$$\pi_J(p) = \{(x \downarrow J) \mid x \in p\}.$$

The reverse operation of projection is *cylindrification*. If p is a relation with arity J, then the *cylinder of p* (erected on I) is

$$\pi^{-1}(p) = \{x \mid (x \downarrow J) \in p \wedge x \in \bigotimes E\},$$

where I and E are as defined in page 61. By lemma 5.1.1, we have equivalently

$$\pi^{-1}(p) = p \bowtie \bigotimes E.$$

We call p the *basis* of $\pi^{-1}(p)$. Cylindrification and projection are related as follows.

Lemma 5.1.2
If p is a relation with arity I and $J \subseteq I$, then $p \subseteq \pi^{-1}(\pi_J(p))$.

Proof: $x \in p \Rightarrow ((x \downarrow J) \in \pi_J(p) \wedge x \in \bigotimes E) \Rightarrow x \in \pi^{-1}(\pi_J(p))$. ■

Lemma 5.1.3

If p is a relation with arity $J \subseteq I$, then $p = \pi_J(\pi^{-1}(p))$.

Proof:

$$\begin{aligned} & x \in \pi_J(\pi^{-1}(p)) \\ \Leftrightarrow\ & \exists y \in \pi^{-1}(p) \text{ s.t. } (y \downarrow J) = x \\ \Leftrightarrow\ & \exists y : x = (y \downarrow J) \in p \wedge y \in \bigotimes E \\ \Leftrightarrow\ & x \in p. \end{aligned}$$

■

We now state some useful properties of projection and cylindrification.

Proposition 5.1.4 [5]

If $J \subseteq K$ and p, q are relations with arity K, then $\pi_J(p \cap q) \subseteq \pi_J(p) \cap \pi_J(q)$. ■

Lemma 5.1.5

If p and q are relations with arity J, then $\pi^{-1}(p \cap q) = \pi^{-1}(p) \cap \pi^{-1}(q)$.

Proof:

$$\begin{aligned} & x \in (\pi^{-1}(p) \cap \pi^{-1}(q)) \\ \Leftrightarrow\ & x \in \pi^{-1}(p) \wedge x \in \pi^{-1}(q) \\ \Leftrightarrow\ & (x \downarrow J) \in p \wedge (x \downarrow J) \in q \wedge x \in \bigotimes E \\ \Leftrightarrow\ & (x \downarrow J) \in (p \cap q) \wedge x \in \bigotimes E \\ \Leftrightarrow\ & x \in \pi^{-1}(p \cap q). \end{aligned}$$

■

Proposition 5.1.6 [5]

If $J \subseteq K$ and $p \subseteq q$ are relations with arity K, then $\pi_J(p) \subseteq \pi_J(q)$. ■

Proposition 5.1.7 [5]

If $p \subseteq q$ are relations with arity J, then $\pi^{-1}(p) \subseteq \pi^{-1}(q)$. ∎

5.2 The Ashby Chain

Ashby [5] describes a method for the analysis of constraint relations. If p is a relation with arity I with $|I| = n > 0$, then the *cylindrical closure of order k* ($k \leq n$) of p is defined by

$$C_k(p) = \begin{cases} \bigotimes E & \text{if } k = 0 \\ \bigcap_{J \subseteq I, |J| = k} \pi^{-1}(\pi_J(p)) & \text{if } k > 0. \end{cases}$$

$C_k(p)$ is formed by (1) taking p's projections on all $J \subseteq I$ with $|J| = k$, (2) forming all cylinders erected on I usir : these projections as bases (page 62), and then (3) taking the intersection of all the cylinders. The following lemma shows that steps (2) and (3) are equivalent to computing the natural join of the $\pi_J(p)$ for each $|J| = k$.

Lemma 5.2.1

If $J \in (K \rightarrow \mathcal{P}(I))$ is a family of subsets of I and $p \in (K \rightarrow \bigcup_{k \in K} \mathcal{P}(\otimes(E \downarrow J_k)))$ is a family of relations such that $p_k \subseteq \otimes(E \downarrow J_k)$ is a relation with arity J_k, then

$$\bigcap_{k \in K} \pi^{-1}(p_k) = (\bowtie p) \bowtie \bigotimes E.$$

Proof:

$$\begin{aligned} & x \in \bigcap_{k \in K} \pi^{-1}(p_k) \\ \Leftrightarrow \quad & x \in \bigcap_{k \in K} (p_k \bowtie \bigotimes E) \\ \Leftrightarrow \quad & \forall k \in K : ((x \downarrow J_k) \in p_k \wedge x \in \bigotimes E) \text{ by lemma 5.1.1} \\ \Leftrightarrow \quad & (\forall k \in K : (x \downarrow J_k) \in p_k) \wedge x \in \bigotimes E \\ \Leftrightarrow \quad & x \in (\bowtie p) \bowtie \bigotimes E. \end{aligned}$$

■

In general, cylindrical closures do not distribute over intersection but the following formula holds.

Lemma 5.2.2

If p and q are relations with arity I and $k \leq n$, then $C_k(p \cap q) \subseteq C_k(p) \cap C_k(q)$.

Proof: The case for $k = 0$ is trivial. If $k > 0$, then

$$\begin{aligned} C_k(p \cap q) &= \bigcap_{J \subseteq I, |J|=k} \pi^{-1}(\pi_J(p \cap q)) \\ &\subseteq \bigcap_{J \subseteq I, |J|=k} \pi^{-1}(\pi_J(p) \cap \pi_J(q)) \text{ by propositions 5.1.4 and 5.1.7} \\ &= (\bigcap_{J \subseteq I, |J|=k} \pi^{-1}(\pi_J(p))) \cap (\bigcap_{|J|=k} \pi^{-1}(\pi_J(q))) \text{ by lemma 5.1.5} \\ &= C_k(p) \cap C_k(q). \end{aligned}$$

■

Cylindrical closures can also be expressed in logic programming. If $n = 4$, the cylindrical closures of p can be expressed by the following Horn clauses:

$$\begin{aligned} &[C_0(p)](X_1, X_2, X_3, X_4). \\ &[C_1(p)](X_1, X_2, X_3, X_4) \leftarrow p(X_1, _, _, _), p(_, X_2, _, _), p(_, _, X_3, _), p(_, _, _, X_4). \\ &[C_2(p)](X_1, X_2, X_3, X_4) \leftarrow p(X_1, X_2, _, _), p(X_1, _, X_3, _), p(X_1, _, _, X_4), \\ &\qquad p(_, X_2, X_3, _), p(_, X_2, _, X_4), p(_, _, X_3, X_4). \\ &[C_3(p)](X_1, X_2, X_3, X_4) \leftarrow p(X_1, X_2, X_3, _), p(X_1, X_2, _, X_4), p(X_1, _, X_3, X_4), \\ &\qquad p(_, X_2, X_3, X_4). \\ &[C_4(p)](X_1, X_2, X_3, X_4) \leftarrow p(X_1, X_2, X_3, X_4). \end{aligned}$$

It may already be apparent to the readers that the cylindrical closures of p shrinks as the order goes up. This behavior is stated in the following proposition.

Proposition 5.2.3 [5]

If $k_1 \geq k_2$, then $C_{k_1}(p) \subseteq C_{k_2}(p)$. ■

A corollary is that the cylindrical closures of a relation p form a chain.

Corollary 5.2.4 (The Ashby Chain) [5]

$\bigotimes E = C_0(p) \supseteq C_1(p) \supseteq \cdots \supseteq C_k(p) \supseteq \cdots \supseteq C_n(p) = p.$ ■

By corollary 5.2.4, the cylindrical closures of order 0 to n of a relation p form a sequence of successive approximations of p. This is best visualized with a geometrical example. Let p be a solid unit sphere in $\mathbb{R}^3$ defined by:

$$p = \{(x, y, z) \mid x^2 + y^2 + z^2 \leq 1\}.$$

The Ashby Chain of p is:

$$\begin{array}{rcl} & C_0(p) & = & \mathbb{R}^3 \\ \supset & C_1(p) & = & \{(x, y, z) \mid x^2 \leq 1 \wedge y^2 \leq 1 \wedge z^2 \leq 1\} \\ \supset & C_2(p) & = & \{(x, y, z) \mid x^2 + y^2 \leq 1 \wedge x^2 + z^2 \leq 1 \wedge y^2 + z^2 \leq 1\} \\ \supset & C_3(p) & = & p. \end{array}$$

$C_0(p)$ is the entire 3-space; $C_1(p)$ is a unit cube centered at the origin with edges parallel to the axes; $C_2(p)$ is the intersection of three cylinders with unit radius, each of which is along an axis; and $C_3(p)$ is the sphere itself.

The definition of cylindrical closure guarantees that $C_n(p) = p$. However, this bound can be reached earlier in the Ashby Chain. The *cylindrance* of p is k if and only if (1) $C_k(p) = p$ and (2) $\forall k'$ such that $C_{k'}(p) = p$, $k' \geq k$. In other words, the cylindrance of a relation p is the smallest order k such that $C_k(p) = p$. An n-ary relation p with arity I of cylindrance k is composed of k-ary relations, which is combinatorially less complex than an n-ary relation. Therefore, cylindrance can be

used as a measure of the intrinsic complexity of a relation. The following theorem helps to determine the cylindrance of a relation.

Theorem 5.2.5 [5]

If p is the intersection of cylinders whose bases are all of k dimensions and fewer, then the cylindrance of p cannot exceed k. ∎

Corollary 5.2.6

Let p and q be relations with arity I. If the cylindrance of p is k' and the cylindrance of q is k'', then the cylindrance of $p \cap q$ cannot exceed $\max(k',k'')$.

Proof: Let $k = \max(k', k'')$. Since k' and k'' are less than or equal to k, we have $C_k(p) = p$ and $C_k(q) = q$ by corollary 5.2.4. Therefore, $p \cap q = C_k(p) \cap C_k(q)$, which is an intersection of cylinders whose bases are of k dimensions. By theorem 5.2.5, the cylindrance of $p \cap q$ cannot exceed k. ∎

Corollary 5.2.7

If $k \geq k'$, then $C_k(C_{k'}(p)) = C_{k'}(p)$.

Proof: By theorem 5.2.5, the cylindrance of $C_{k'}(p)$ cannot exceed k'. Therefore, $C_k(C_{k'}(p)) = C_{k'}(p)$. ∎

5.3 The Generalized Ashby Chain

A relation p with arity I of cylindrance k is composed of the intersection of *all* projections of p on $J \subseteq I$ of size k. This definition is quite restrictive. Sometimes it is unnecessary to include all subsets J of I of certain size to reach p as shown in the following example. We consider the set of 5-letter palindromes such that the middle

letter is always an "**a**." Strings, such as "**cdadc**" and "**zzazz**," are elements of this set. We can represent the set by the following relation:

$$p = \{x \in (I \to \Sigma) \mid x_1 = x_5, x_2 = x_4, x_3 = \mathbf{a}\},$$

where $I = \{1,2,3,4,5\}$ and Σ is the English alphabet. Let $\mathcal{S} = \{\{1,5\},\{2,4\},\{3\}\}$. It is easy to verify that

$$\begin{aligned}
\pi_{\{1,5\}}(p) &= \{x \in (\{1,5\} \to \Sigma) \mid x_1 = x_5\} \\
\pi^{-1}(\pi_{\{1,5\}}(p)) &= \{x \in (I \to \Sigma) \mid x_j \in \Sigma \text{ if } j \in \{2,3,4\}, x_1 = x_5\} \\
\pi_{\{2,4\}}(p) &= \{x \in (\{2,4\} \to \Sigma) \mid x_2 = x_4\} \\
\pi^{-1}(\pi_{\{2,4\}}(p)) &= \{x \in (I \to \Sigma) \mid x_j \in \Sigma \text{ if } j \in \{1,3,5\}, x_2 = x_4\} \\
\pi_{\{3\}}(p) &= \{x \in (\{3\} \to \Sigma) \mid x_3 = \mathbf{a}\} \\
\pi^{-1}(\pi_{\{3\}}(p)) &= \{x \in (I \to \Sigma) \mid x_j \in \Sigma \text{ if } j \in \{1,2,4,5\}, x_3 = \mathbf{a}\}.
\end{aligned}$$

Therefore,

$$\bigcap_{J \in \mathcal{S}} \pi^{-1}(\pi_J(p)) = p.$$

$\mathcal{S}$ contains a mixture of subsets of size 2 and 1. This example suggests the following generalization of cylindrical closure. If p is a relation with arity I, then the *generalized cylindrical closure of order* $\mathcal{S}$ ($\mathcal{S} \subseteq \mathcal{P}(I)$) of p is

$$C_{\mathcal{S}}(p) = \begin{cases} \bigotimes E & \text{if } \mathcal{S} = \emptyset \\ \bigcap_{J \in \mathcal{S}} \pi^{-1}(\pi_J(p)) & \text{otherwise} . \end{cases}$$

Theorem 5.3.1

If $\mathcal{S}$ and $\mathcal{T}$ are non-empty subsets of $\mathcal{P}(I)$ and $\mathcal{S} \subseteq \mathcal{T}$, then $C_{\mathcal{T}}(p) \subseteq C_{\mathcal{S}}(p)$.

Proof: Let $\mathcal{D} = \mathcal{T} - \mathcal{S}$. We have

$$C_{\mathcal{T}}(p) = \bigcap_{J \in \mathcal{T}} \pi^{-1}(\pi_J(p)) = (\bigcap_{J \in \mathcal{S}} \pi^{-1}(\pi_J(p)) \cap \bigcap_{J \in \mathcal{D}} \pi^{-1}(\pi_J(p))) \subseteq \bigcap_{J \in \mathcal{S}} \pi^{-1}(\pi_J(p)).$$

■

Let K be a totally ordered set with ordering relation $<$. Suppose α and ω are the smallest and largest elements of K with respect to $<$. We consider a family $\mathcal{S} \in (K \rightarrow \mathcal{P}(\mathcal{P}(I)))$ in subsets of $\mathcal{P}(I)$ such that (1) $\mathcal{S}_\alpha = \emptyset$, (2) $\mathcal{S}_\omega = \mathcal{P}(I)$, and

$$(3)\ (a \in K \wedge b \in K \wedge a < b) \Rightarrow \mathcal{S}_a \subseteq \mathcal{S}_b.$$

Therefore $\mathcal{S}$ is a chain of subsets of $\mathcal{P}(I)$. The following is a corollary from theorem 5.3.1.

Corollary 5.3.2 (The Generalized Ashby Chain)
If $k \in K$, then $\bigotimes E = C_{\mathcal{S}_\alpha}(p) \supseteq \cdots \supseteq C_{\mathcal{S}_k}(p) \supseteq \cdots \supseteq C_{\mathcal{S}_\omega}(p) = p.$ ■

The *generalized cylindrance* of p with respect to $\mathcal{S}$ is the smallest $\mathcal{S}_k$ ($k \in K$) such that $C_{\mathcal{S}_k}(p) = p$. We observe, by proposition 5.2.3, the following equalities:

$$C_k(p) = \bigcap_{i=0}^{k} C_i(p) = \begin{cases} \bigotimes E & \text{if } k = 0 \\ \bigcap_{J \subseteq I, |J| \leq k} \pi^{-1}(\pi_J(p)) & \text{if } k > 0 \end{cases}$$

If we define $\mathcal{S}_k = \{J \mid J \subseteq I \wedge |J| \leq k\}$, then $C_k(p) = C_{\mathcal{S}_k}(p)$. Therefore, the Ashby Chain is a special case of the Generalized Ashby Chain.

5.4 The General Scheme

We show that the proper choice of projections, on which cylinders are erected, depends on the structure of the relations. The generalized cylindrical closures facilitate this flexibility in the choice of projection. This forms the basis of our general framework for the domain restriction operations. Given a constraint c in the form of an n-ary relation with arity I, we choose a collection $\mathcal{S}$ of subsets of I. Each element

of $\mathcal{S}$ is called a ***projection index set***. We are interested in the circumscribing set c' constructed as follows:

$$c' = C_{\mathcal{S}}(c) = \bigcap_{J \in \mathcal{S}} \pi^{-1}(\pi_J(c)).$$

By theorem 5.3.1, $c \subseteq c'$. Let $J \in (K \to \mathcal{P}(I))$. Suppose $p \in (K \to \bigcup_{k \in K} \mathcal{P}(\bigotimes(E \downarrow J_k)))$ is a family of relations such that p_k is a relation with arity J_k.

$$Q = \bigcap_{k \in K} \pi^{-1}(p_k) \Rightarrow \pi_J(Q) \subseteq p_k \text{ for each } k \in K.$$

However, c' has a closer relationship with $\pi_J(c)$ as shown in the following lemma.

Lemma 5.4.1

If $J \in \mathcal{S}$, then $\pi_J(c') = \pi_J(c)$.

Proof: We have $c \subseteq c'$. By lemma 5.1.6, $\pi_J(c) \subseteq \pi_J(c')$.

Let $y \in \pi_J(c')$. There exists $x \in c'$ such that $y = (x \downarrow J)$. Thus $x \in \pi^{-1}(\pi_J(c))$ and $(x \downarrow J) = y \in \pi_J(c)$. Therefore, $\pi_J(c') \subseteq \pi_J(c)$. ∎

For all $J \in \mathcal{S}$, each $\pi_J(c)$ is called the *cylindric basis* of c'. Note that $\bigcup \mathcal{S}$ may not be I and thus $\bigcap_{J \in \mathcal{S}} \pi^{-1}(\pi_J(c)) \neq \bowtie_{J \in \mathcal{S}} \pi_J(c)$ in general.

The next step is to determine the membership of a tuple in c'. The structure of c', an intersection of cylinders, allows us to perform the test without computing c' explicitly. We only need to compute the cylindric bases $\pi_J(c)$.

Lemma 5.4.2

If x is an I-tuple, then $x \in c' \Leftrightarrow (\forall J \in \mathcal{S} : (x \downarrow J) \in \pi_J(c))$.

Proof:

$$\begin{aligned} & x \in c' \\ \Leftrightarrow\ & x \in \bigcap_{J \in \mathcal{S}} \pi^{-1}(\pi_J(c)) \\ \Leftrightarrow\ & \forall J \in \mathcal{S} : x \in \pi^{-1}(\pi_J(c)) \\ \Leftrightarrow\ & \forall J \in \mathcal{S} : (x \downarrow J) \in \pi_J(c). \end{aligned}$$

■

Since our method aims to prune the set of possible candidate solutions, we are more interested in checking whether a tuple is *not* a member of c'. The following corollary is more suitable for this purpose.

Corollary 5.4.3
If x is an I-tuple, then $x \notin c' \Leftrightarrow \exists J \in \mathcal{S}$ s.t. $(x \downarrow J) \notin \pi_J(c)$.

Proof: From lemma 5.4.2. ■

Corollary 5.4.3 suggests more than just deciding the membership of a tuple in c': if a tuple fails the membership test because its projection onto indices in J does not belong to $\pi_J(c)$, then we can discard all tuples with the same projection.

5.5 Instances Of The General Scheme

We obtain instances of the general scheme by choosing different collections $\mathcal{S}$ of projection index sets. In this section, we show how the work of some of our predecessors is classified as instances of our scheme.

Let M be an index set of size m. The specification of a constraint satisfaction problem (CSP) consists of a family X of m variables with indices from M, a family D of domains of possible values for each variable, and a set of predicates that constrain

the values of the variables. A domain D_k, where $k \in M$, is a set of values. Note that we can construct a corresponding unary relation from D_k, and:

$$x \in \bigotimes(D \downarrow \{k\}) \Leftrightarrow x_k \in D_k.$$

We abuse notation by denoting this unary relation by D_k as well. Mackworth's CSP formulation [46] considers only unary and binary predicates. Let u be a family of unary predicates with index set M and b a family of binary predicates with index set $M \times M$. Each predicate denotes a relation. We overload u_k and b_{jk} to name the relations denoted by the predicates u_k and b_{jk} respectively. Furthermore, we use the convention that $\forall x : u_k(x) \Leftrightarrow b_{kk}(x, x)$ and $\forall x, y : b_{jk}(x, y) \Leftrightarrow b_{kj}(y, x)$. A CSP is specified by the following formula:

$$\forall a \in M, \exists X_a \in D_a \text{ s.t. } \bigwedge_{k \in M} u_k(X_k) \wedge \bigwedge_{k \in M, j \in M} b_{jk}(X_j, X_k).$$

Although, in general, we may not have explicit constraints between all pairs of variables X_k and X_j in an application, we can assume that it exists; in this case, b_{kj} is the relation $D_k \bowtie D_j$.

We can view the CSP specification as a *network*, which is a complete and labeled graph. Each variable X_k corresponds to the *node* k; each node k is associated with a *label* which is the domain D_k. Each relation b_{kj} corresponds to the arc (k, j) between nodes k and j. A *path* $(k_0, k_1, \ldots, k_j)$ of length j is a sequence of nodes. Please note that the nodes in a path are not necessarily distinct. Thus a path may contain cycles.

Fikes [23], Waltz [78], and Montanari [52] proposed incremental techniques that remove inconsistencies from the network and thus reduce the search space. Mackworth recognized the common characteristics of their algorithms and put them in the uniform framework of constraint satisfaction algorithms. Mackworth's formulation leads to three notions of consistency:

- Node k is ***node consistent*** if and only if for any value $x_k \in D_k$, $u_k(x_k)$ holds.
- Arc (k, j) is ***arc consistent*** if and only if for any value $x_k \in D_k$, there is a value $x_j \in D_j$ such that $b_{kj}(x_k, x_j)$ holds.
- A path through the nodes $(k_0, k_1, \ldots, k_j)$ is ***path consistent*** if and only if for any values $x_0 \in D_{k_0}$ and $x_j \in D_{k_j}$ such that $b_{k_0 k_j}(x_0, x_j)$ hold, there is a sequence of values $x_1 \in D_{k_1}, \ldots, x_{j-1} \in D_{k_{j-1}}$ such that $b_{k_0 k_1}(x_0, x_1)$, $b_{k_1 k_2}(x_1, x_2)$, ..., and $b_{k_{j-1} k_j}(x_{j-1}, x_j)$ hold.

A network is *node* (or *arc* or *path*) *consistent* if every node (or arc or path) of its graph is consistent. Two networks N_1 and N_2, each with m nodes, are ***equivalent*** if and only if the set of tuples satisfying N_1 is identical to the set of tuples satisfying N_2. Mackworth presents node, arc, and path *consistency algorithms* [46] that reduce a network to an equivalent and, correspondingly, node, arc, and path consistent network.

Each consistency algorithm consists of two main components: a ***domain restriction operation*** and a ***symbolic relaxation algorithm***, which coordinates the application of the domain restriction operation on constraints in a network. We are interested in the domain restriction operations, which are responsible for achieving the corresponding node, arc, and path consistencies.

Node consistency. For each node k, we are interested in 1-tuples, each of which satisfies u_k and is an element of D_k. Therefore, the constraint relation is $c = u_k \cap D_k$ with arity $I = \{k\}$. The domain restriction operation updates D_k to D'_k by

$$D'_k = D_k \cap \{x_k \mid u_k(x_k) \text{ holds}\},$$

which is equivalent to

$$D'_k = D_k \cap u_k = c.$$

The collection $\mathcal{S}$ of projection index sets is trivial: $\mathcal{S} = \{\{k\}\}$. The following theorem shows that the circumscribing set c' is c.

Theorem 5.5.1

$c' = D'_k = c$.

Proof: $c' = \bigcap_{J \in \mathcal{S}} \pi^{-1}(\pi_J(c))) = \pi^{-1}(\pi_{\{k\}}(D_k \cap u_k)) = D_k \cap u_k = D'_k = c$. ∎

Arc consistency. Every two distinct nodes k and j are associated with two arcs (k, j) and (j, k). The arcs always exist because if there is no explicit constraint between variables X_k and X_j, we can assume that the constraint b_{kj} is $D_k \bowtie D_j$ (page 72). The constraint relation is $c = b_{kj} \cap (D_k \bowtie D_j)$, which is a binary relation, with arity $I = \{k, j\}$. The domain restriction operations for the arcs are:

$$\begin{aligned} D'_k &= D_k \cap \{x_k \mid \exists x_j \in D_j \text{ s.t. } b_{kj}(x_k, x_j)\} \\ D'_j &= D_j \cap \{x_j \mid \exists x_k \in D_k \text{ s.t. } b_{kj}(x_k, x_j)\}. \end{aligned}$$

Lemma 5.5.2

$D'_k = \pi_{\{k\}}(c)$ and $D'_j = \pi_{\{j\}}(c)$.

Proof: We prove the first equality. The proof of the second one is similar.

$$\begin{aligned} D'_k &= D_k \cap \{x_k \mid \exists x_j \in D_j \text{ s.t. } b_{kj}(x_k, x_j)\} \\ &= \{x_k \mid x_k \in D_k \wedge x_j \in D_j \wedge b_{kj}(x_k, x_j)\} \\ &= \{(x \downarrow \{k\}) \mid x \in D_k \bowtie D_j \wedge x \in b_{kj}\}^1 \\ &= \{(x \downarrow \{k\}) \mid x \in (D_k \bowtie D_j) \cap b_{kj}\} \\ &= \pi_{\{k\}}((D_k \bowtie D_j) \cap b_{kj}). \end{aligned}$$

∎

[1] We abuse notation by equating a set of constants to the unary relation induced by this set, as explained in page 72.

Therefore, the domain restriction operations compute $D'_k \bowtie D'_j$. We choose $\mathcal{S} = \{\{k\}, \{j\}\}$. The following theorem shows that the domain restriction operations and our scheme achieve the same pruning.

Theorem 5.5.3

$c' = \bigcap_{J \in \mathcal{S}} \pi^{-1}(\pi_J(c)) = D'_k \bowtie D'_j$.

Proof:

$$
\begin{aligned}
c' &= \bigcap_{J \in \mathcal{S}} \pi^{-1}(\pi_J(c)) \\
&= \pi^{-1}(\pi_{\{k\}}(c)) \cap \pi^{-1}(\pi_{\{j\}}(c)) \\
&= \pi^{-1}(D'_k) \cap \pi^{-1}(D'_j) \text{ by lemma 5.5.2} \\
&= D'_k \bowtie D'_j \text{ by lemma 5.2.1}
\end{aligned}
$$

■

We can easily extend this result to hyper-arc consistency [47, 53], which is used in constraint logic programming systems such as [74, 41, 42]. In this framework, n-ary constraints are considered as hyper-arcs in a hyper-graph, which corresponds to the constraint network. A constraint p with arity I with domains D is *hyper-arc consistent* if and only if for all $x_i \in D_i$, there exist $x_j \in D_j$ for all $j \in I - \{i\}$ such that $x \in p$ holds. In that case, $c = p \cap \bowtie D$. For each $i \in I$, the domain restriction operation is

$$D'_i = D_i \cap \{x_i \mid \exists x_j \in D_j (j \in I - \{i\}) \text{ s.t. } x \in p\}.$$

We choose $\mathcal{S} = \{\{1\}, \dots, \{n\}\}$. A similar result as theorem 5.5.3 holds.

Theorem 5.5.4

$c' = \bigcap_{J \in \mathcal{S}} \pi^{-1}(\pi_J(c)) = \bowtie D'$. ■

Path Consistency. The following theorem due to Montanari suggests that we only need to consider paths of length 2 in a path consistent algorithm.

Theorem 5.5.5 [52]
If every path of length 2 of a network with a complete graph is path consistent, then the network is path consistent. ■

Every three distinct nodes j, k, and l are associated with three paths[2] (j,k,l), (j,l,k), and (k,j,l) of length 2. We are interested in 3-tuples, each of which is an element of $D_j \bowtie D_k \bowtie D_l$ and satisfies b_{jk}, b_{jl}, and b_{kl}. Therefore the constraint relation is $c = (b_{jk} \bowtie b_{jl} \bowtie b_{kl}) \cap (D_j \bowtie D_k \bowtie D_l)$ with arity $I = \{i,j,k\}$. The domain restriction operations for the paths are:

$$\begin{aligned}
D'_{jl} &= (D_j \bowtie D_l) \cap b_{jl} \cap \{(x \downarrow \{j,l\}) \mid x_k \in D_k \wedge (x \downarrow \{j,k\}) \in b_{jk} \wedge (x \downarrow \{k,l\}) \in b_{kl}\} \\
D'_{jk} &= (D_j \bowtie D_k) \cap b_{jk} \cap \{(x \downarrow \{j,k\}) \mid x_l \in D_l \wedge (x \downarrow \{j,l\}) \in b_{jl} \wedge (x \downarrow \{k,l\}) \in b_{kl}\} \\
D'_{kl} &= (D_k \bowtie D_l) \cap b_{kl} \cap \{(x \downarrow \{k,l\}) \mid x_j \in D_j \wedge (x \downarrow \{k,j\}) \in b_{kj} \wedge (x \downarrow \{j,l\}) \in b_{jl}\}
\end{aligned}$$

Lemma 5.5.6
$D'_{jl} = \pi_{\{j,l\}}(c)$, $D'_{jk} = \pi_{\{j,k\}}(c)$, and $D'_{kl} = \pi_{\{k,l\}}(c)$.

Proof: We prove the first equality. The proofs of the other two are similar.

$$\begin{aligned}
D'_{jl} &= (D_j \bowtie D_l) \cap b_{jl} \cap \{(x \downarrow \{j,l\}) \mid x_k \in D_k \wedge (x \downarrow \{j,k\}) \in b_{jk} \wedge (x \downarrow \{k,l\}) \in b_{kl}\} \\
&= \pi_{\{j,l\}}(\{(x \downarrow \{j,k,l\}) \mid x_j \in D_j \wedge x_k \in D_k \wedge x_l \in D_l \wedge (x \downarrow \{j,l\}) \in b_{jl} \wedge \\
&\qquad (x \downarrow \{j,k\}) \in b_{jk} \wedge (x \downarrow \{k,l\}) \in b_{kl}\}) \\
&= \pi_{\{j,l\}}(\{x \mid x \in ((b_{jk} \bowtie b_{jl} \bowtie b_{kl}) \cap (D_j \bowtie D_k \bowtie D_l))\} \\
&= \pi_{\{j,l\}}(c).
\end{aligned}$$

[2]Paths (l,k,j), (k,l,j), and (l,j,k) are redundant since predicates, say, b_{jk} and b_{kj} denote the same relation.

■

Therefore, the domain restriction operations compute $D'_{jl} \bowtie D'_{jk} \bowtie D'_{kl}$. We choose $\mathcal{S} = \{\{j,k\},\{j,l\},\{k,l\}\}$. The following theorem shows that the domain restriction operations and our scheme achieve the same pruning.

Theorem 5.5.7
$c' = \bigcap_{J \in \mathcal{S}} \pi^{-1}(\pi_J(c)) = \bigcap_{J \in \mathcal{S}} \pi^{-1}(D'_J) = \bowtie D'$.

Proof:

$$\begin{aligned} c' &= \bigcap_{J \in \mathcal{S}} \pi^{-1}(\pi_J(c)) \\ &= \pi^{-1}(D'_{jl}) \cap \pi^{-1}(D'_{jk}) \cap \pi^{-1}(D'_{kl}) \text{ by lemma 5.5.6} \\ &= \bowtie D' \text{ by lemma 5.2.1.} \end{aligned}$$

■

Chapter 6

Concluding Remarks

6.1 Summary And Contributions

Relational Interval Arithmetic. We have specified the essential components of a relational interval arithmetic system, consisting of an interval narrowing operation and a relaxation algorithm. The interval narrowing operation is applicable to any arithmetic relations, satisfying a simple criterion. The operation uses the n set functions $F_i(p)$ associated with each relation $p \subseteq \mathbb{R}^n$. Our criterion is that each $F_i(p)$ maps from real interval(s) to a real interval. We check that the usual arithmetic relations, possibly by suitable partitioning, satisfy this criterion. They can be used as primitives in a relational interval arithmetic system. Algorithm 2.1 (page 39) coordinates the applications of interval narrowing to transform a constraint network into an equivalent and stable form. We prove properties of relational interval arithmetic. Theorem 2.3.1 (page 26) is a *soundness* and *completeness* result for interval narrowing: an answer is in the input intervals if and only if it is in the output intervals. Interval narrowing is an idempotent operation as shown in lemma 2.5.1 (page 37).

Theorem 2.5.2 (page 40) guarantees the termination of algorithm 2.1. Most importantly, we do not assume ideal real-number arithmetic in digital hardware. The effect of outward-rounding plays a significant role in our results.

Constraint Logic Programming. Theorems 3.2.1 (page 46) and 4.1.1 (page 55) formulate interval narrowing as logical inference rules. They form the basis of incorporating relational interval arithmetic into the constraint logic programming languages CHIP and CLP($\mathcal{R}$). The enhanced languages, ICHIP and ICLP($\mathcal{R}$), are general-purpose programming languages allowing compact description of constraint networks. Consequently, we can describe programs, constraints, queries, intervals, answers, and variables in a coherent and semantically precise language—*logic*. ICHIP has the same semantics as CHIP; and ICLP($\mathcal{R}$) is based on the CLP scheme extended with a constraint simplification step. Computations in the extended languages are logical inferences; thus *numerical computations are deductions* in our framework. Furthermore, our proposed languages represent a departure from conventional numerical computing, *offering a relational and declarative method of numerical programming yet to be explored*. To show the feasibility of our method, we have specified and implemented an ICLP($\mathcal{R}$) prototype in CLP($\mathcal{R}$).

Constraint Satisfaction Techniques. Relational interval arithmetic is a form of constraint satisfaction. We make a connection between constraint satisfaction techniques and Ashby's notion of cylindrance. We generalize the notion of cylindrical closures to obtain the Generalized Ashby Chain as shown in theorem 5.3.1 (page 68) and corollary 5.3.2 (page 69). This forms the basis of a general framework (page 69) for the domain restriction operations of constraint satisfaction algorithms. We show that some of our predecessors' algorithms are instances of our scheme in theorems 5.5.1

(page 74), 5.5.3 (page 75), 5.5.4 (page 75), and 5.5.7 (page 77).

6.2 Suggestions For Further Work

Efficient Implementation. The ICLP($\mathcal{R}$) prototype is an interpreter written in CLP($\mathcal{R}$) but it would be more efficient if it were re-implemented at the source level of CLP($\mathcal{R}$). An implementation is also needed for ICHIP. There is potential parallelism in algorithm 2.1. We expect Leung's doctoral dissertation [44] to be the basis of a parallel or distributed implementation of algorithm 2.1.

Improving the relaxation algorithm. The efficiency of a relational interval arithmetic system depends critically on algorithm 2.1. As described, our relaxation algorithm is similar to Mackworth's AC-3 and the Waltz algorithm. Techniques, such as those proposed in [51, 21], may be applicable to our relaxation algorithm to improve its efficiency. Another approach is to use heuristics to order interval constraints in the active list A (page 39). A useful heuristic is the first-fail principle [25], which states: *to succeed, try first where you are the most likely to fail.*

Complexity Analysis The complexity of an interval narrowing operation is easy to determine since it depends only on the number of floating-point operations performed by each $F_i(p)$ function. It is difficult, however, to formulate a complexity measure for algorithm 2.1 since its termination depends on the precision of the underlying floating-point system. Such a measure must take the precision into account. A starting point for this work is to identify and study some simple classes of interval networks.

Variable identity. Suppose X is the interval $[-1, 2]$. The result of the interval expression $X \otimes X$ is $[-2, 4]$. Since $X \times X$ in real arithmetic is the squaring operation, some readers may have anticipated the value of the interval expression to be in the positive region of the real line. The problem is that interval arithmetic simply multiplies $[-1, 2]$ by $[-1, 2]$ without realizing that the same variable X is used as both the multiplier and the multiplicand, resulting in a larger than desirable interval. We call this the *variable identity* problem of interval arithmetic. To obtain $[1, 4]$ as the answer, we need to use the interval squaring operation $\mathrm{SQR}(X)$, where $\mathrm{SQR}(X) = \{x^2 \mid x \in X\}$.

This variable identity problem also occurs in interval narrowing, so that the output intervals are larger than expected. The culprit is multiple occurrences of a variable in an interval constraint. A solution is based on the following observation, using the `multiply` relation as an example:

$$\forall X, Y : (X, X, Y) \in \mathtt{multiply} \Leftrightarrow (X, Y) \in \mathtt{square},$$
$$\forall X, Y : (X, Y, X), (Y, X, X) \in \mathtt{multiply} \Leftrightarrow (X, Y) \in \mathtt{zeroOrOne},$$
$$\forall X : (X, X, X) \in \mathtt{multiply} \Leftrightarrow (X) \in \mathtt{zeroAndOne},$$

where **square** $= \{(x, y) \mid y = x^2\}$, **zeroOrOne** $= \{(x, y) \mid x = 0 \text{ or } y = 1\}$, and **zeroAndOne** $= \{(0), (1)\}$. When we encounter a **multiply** constraint at runtime, we select an equivalent relation using the instantiation pattern of the variables. For example, the constraint $(\langle X, X, Y\rangle, \mathtt{multiply})$ becomes $(\langle X, Y\rangle, \mathtt{square})$. This runtime optimization will help us to obtain more accurate output intervals.

Prolog III and CAL. Prolog III and CAL refrain from floating-point arithmetic in order to ensure soundness of their numerical computations. It would be interesting to see if the two constraint logic programming languages can be extended with relational interval arithmetic.

Applications. We have tried ICLP($\mathcal{R}$) on the examples in [61, 59, 18, 31]. More experience is needed to test the applicability of ICHIP and ICLP($\mathcal{R}$) on real-life problems and to experiment with the new programming paradigm offered by the languages.

Bibliography

[1] A. Aiba, K. Sakai, Y. Sato, D.J. Hawley, and R. Hasegawa. Constraint logic programming language CAL. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 263–276, Tokyo, Japan, November–December 1988. Ohmsha, Ltd.

[2] G. Alefeld and J. Herzberger. *Introduction to Interval Computations.* Academic Press, New York, 1983. Translated by J. Rokne from *Elemente der Intervallrechnung*.

[3] J. Andrews. *Trilogy: User's Manual.* Complete Logic Systems, 1987.

[4] K.R. Apt and M.H. van Emden. Contributions to the theory of logic programming. *Journal of ACM*, 29(3):841–862, July 1982.

[5] W.R. Ashby. Constraint analysis of many-dimensional relations. *General Systems, Yearbook of the Society for General Systems Research*, 9:99–106, 1964.

[6] E. Bishop. *Foundations of Constructive Analysis.* McGraw-Hill, 1967.

[7] H-J. Boehm, R. Cartwright, M.J. O'Donnell, and M. Riggle. Exact real arithmetic: A case study in higher order programming. In *Proceedings of the 1986 ACM Symposium on Lisp and Functional Programming*, pages 162–173, 1986.

[8] G.S. Boolos and R.C. Jeffrey. *Computability and Logic*. Cambridge University Press, 1980.

[9] D.S. Bridges. *Constructive Functional Analysis*. Pitman, 1979.

[10] B. Buchberger. Gröbner bases: an algebraic method in polynomial ideal theory. Technical report, CAMP-LINZ, 1983.

[11] A. Bundy. A generalized interval package and its use for semantic checking. *ACM Transactions on Mathematical Systems*, 10(4):397–409, 1984.

[12] M. Carlsson and J. Widen. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, 1988.

[13] B.W. Char, G.J. Fee, K.O. Geddes, G.H. Gonnet, and M.B. Monagan. A tutorial introduction to Maple. *Journal of Symbolic Computation*, 2(2):179–200, June 1986.

[14] M.H.M. Cheng, M.H. van Emden, and P.A. Strooper. Complete sets of frontiers in logic-based program transformation. In H. Abramson and M.H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 283–297. MIT Press, 1989. Also as Technical Report DCS-74-IR (LP-3), University of Victoria, Victoria, Canada, April, 1988.

[15] J.G. Cleary. Logical arithmetic. *Future Computing Systems*, 2(2):125–149, 1987.

[16] G.E. Collins. Quantifier elimination for real closed fields: a guide to the literature. In B. Buchberger, Loos R., and G.E. Collins, editors, *Computer Algebra: Symbolic and Algebraic Computation, Computing Supplement #4*, pages 79–81. Springer-Verlag, 1982.

[17] A. Colmerauer. An introduction to Prolog III. *Communications of the ACM*, pages 69–90, July 1990.

[18] E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32:281–331, 1987.

[19] J. de Kleer and J.S. Brown. A qualitative physics based on confluences. *Artificial Intelligence*, 24:7–83, 1984.

[20] T. Dean. Temporal imagery: An approach to reasoning about time for planning and problem solving. Technical Report 433, Yale University, New Haven, CT, USA, 1985.

[21] Y. Deville and P. Van Hentenryck. An efficient arc consistency algorithm for a class of CSP algorithm. In *Proceedings of IJCAI 1991*, pages 325–330, 1991.

[22] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, pages 693–702, Tokyo, Japan, December 1988.

[23] R.E. Fikes. *A Heuristic Program for Solving Problems Stated as Non-deterministic Procedures.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, U.S.A., 1968.

[24] P.R. Halmos. *Naive Set Theory.* Springer-Verlag, 1974.

[25] R.M. Haralick and G.L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[26] W.S. Havens, S. Sidebottom, G. Sidebottom, J. Jones, M. Cuperman, and R. Davison. Echidna constraint reasoning system: Next-generation expert system technology. Technical Report CSS-IS TR 90-09, Centre for Systems Science, Simon Fraser University, Burnaby, B.C., Canada, 1990.

[27] P.J. Hayes. Computation and deduction. In *Proceedings of the Second MFCS Symposium*, pages 105–118. Czechoslovak Academy of Sciences, 1973.

[28] N. Heintze, J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. *The CLP($\mathcal{R}$) Programmer's Manual Version 1.1.* IBM Thomas J Watson Research Center, 1991.

[29] N.C. Heintze, S. Michaylov, P.J. Stuckey, and R. Yap. On meta-programming in CLP($\Re$). In E.L. Lusk and R.A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming 1989*, pages 52–68, 1989.

[30] R.A. Hummel and S.W. Zucker. On the foundations of relaxation labeling processes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(3):267–287, 1983.

[31] E. Hyvönen. Constraint reasoning based on interval arithmetic. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 1193–1198, Detroit, USA, 1989.

[32] J. Jaffar and J-L. Lassez. Constraint logic programming. Technical Report 72, Department of Computer Science, Monash University, Clayton, Victoria, Australia, 1986.

[33] J. Jaffar and J-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM POPL Conference*, pages 111–119, Munich, January 1987.

[34] J. Jaffar, M.J. Maher, P.J. Stuckey, and R.H.C. Yap. Output in CLP($\mathcal{R}$). In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992 (to appear)*, Tokyo, Japan, June 1992.

[35] J. Jaffar and S. Michaylov. Methodology and implementation of a CLP system. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 196–218, Melbourne, Australia, May 1987.

[36] J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap. The CLP($\mathcal{R}$) language and system. Technical Report CMU-CS-90-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 1990.

[37] W. Kahan. A survey of error analysis. In *Information Processing 71*, pages 1214–1239. North Holland, 1972.

[38] R. Kirchner and U. Kulisch. Arithmetic for vector processors. In R.E. Moore, editor, *Reliability in Computing—The Role of Interval Methods in Scientific Computing*, pages 3–42. Academic Press, 1988.

[39] R.A. Kowalski. Predicate logic as a programming language. In *Proceedings of IFIP 74*, pages 569-574, Stockholm, 1974. North Holland.

[40] R.A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, August 1979.

[41] J.H.M. Lee and M.H. van Emden. Numerical computation can be deduction in CHIP. Technical Report LP-19 (DCS-184-IR), Department of Computer Science, University of Victoria, Victoria, B.C., Canada, December 1991. (submitted for publication).

[42] J.H.M. Lee and M.H. van Emden. Adapting CLP($\mathcal{R}$) to floating-point arithmetic. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 996–1003, Tokyo, Japan, June 1992. Also as Technical Report DCS-183-IR (LP-18), University of Victoria, Victoria, Canada, 1991.

[43] V.A. Lee Jr. and H-J. Boehm. Optimizing programs over the constructive reals. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 102–111, White Plains, New York, USA, June 1990.

[44] H.F. Leung. *Distributed Constraint Logic Programming.* PhD thesis, Imperial College of Science, Technology and Medicine, London, United Kingdom, 1992.

[45] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[46] A.K. Mackworth. Consistency in networks of relations. *AI Journal*, 8(1):99–118, 1977.

[47] A.K. Mackworth. On reading sketch maps. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 598–606, Cambridge, MA, U.S.A., 1977.

[48] A.K. Mackworth, J.A. Mulder, and W.S. Havens. Hierarchical arc consistency: Exploiting structured domains in constraint satisfaction problems. *Computational Intelligence*, 1:118–126, 1985.

[49] M.J. Maher and P.J. Stuckey. Expanding query power in contraint logic programming languages. In *Proceedings of the North American Conference on Logic Programming 1989*, pages 20–36, Cleveland, Ohio, U.S.A., 1989.

[50] D.V. McDermott and E. Davis. Planning routes through uncertain territory. *Artificial Intelligence*, 22:107–156, 1984.

[51] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[52] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7(2):95–132, 1974.

[53] U. Montanari and F. Rossi. Fundamental properties of networks of constraints: A new formulation. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, pages 426–449. Springer Series in Symbolic Computation, 1988.

[54] R.E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[55] R.E. Moore, editor. *Reliability in Computing—The Role of Interval Methods in Scientific Computing*. Academic Press, 1988.

[56] J.L. Morris. *Computational Methods in Elementary Numerical Analysis*. John Wiley & Sons, 1983.

[57] Members of the Binary Floating-point Arithmetic Working Group. IEEE standard for binary floating-point arithmetic. Technical Report ANSI/IEEE Std 754-1985, The Institute of Electrical and Electronics Engineers, New York, USA, 1987.

[58] Members of the Radix-Independent Floating-point Arithmetic Working Group. IEEE standard for radix-independent floating-point arithmetic. Technical Report ANSI/IEEE Std 854-1987, The Institute of Electrical and Electronics Engineers, New York, USA, 1987.

[59] W. Older and A. Vellino. Extending Prolog with constraint arithmetics on real intervals. In *Proceedings of the Canadian Conference on Computer & Electrical Engineering*, Ottawa, Canada, 1990.

[60] W. Older and A. Vellino. Constraint arithmetic on real intervals. In A. Colmerauer and F. Benhamou, editors, *Constraint Logic Programming: Selected Research*. MIT Press, 1992.

[61] W.J. Older. Interval arithmetic specification. Research Report 89032, Computing Research Laboratory, Bell-Northern Research, Ottawa, Ont., Canada, July 198).

[62] W. Rudin. *Principles of Mathematical Analysis*. McGraw Hill, third edition, 1976.

[63] S.M. Rump. Algorithms for verified inclusions theory and practice. In R.E. Moore, editor, *Reliability in Computing—The Role of Interval Methods in Scientific Computing*, pages 109–126. Academic Press, 1988.

[64] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley and Sons, 1986.

[65] G. Sidebottom and W.S. Havens. Hierarchical arc consistency applied to numeric constraint processing in logic programming. Technical Report CSS-IS TR 91-06, Centre for Systems Science, Simon Fraser University, Burnaby, B.C., Canada, 1991.

[66] G. Sidebottom and W.S. Havens. Hierarchical arc consistency for disjoint real intervals in constraint logic programming. *Computational Intelligence*, 1992. (To appear).

[67] R.V. Southwell. *Relaxation Methods in Theoretical Physics*. Oxford University Press, 1946.

[68] P.J. Stuckey. *On the Foundations of Constraint Logic Programming.* PhD thesis, Monash University, Victoria, Australia, October 1987.

[69] I.E. Sutherland. *SKETCHPAD: a Man-Machine Graphical Communication System.* PhD thesis, MIT Lincoln Labs, Cambridge, MA, 1963.

[70] A. Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, second edition, 1951.

[71] S. Ullman. Relaxation and constrained optimization by local processes. *Computer Graphics and Image Processing*, 10:115–125, 1979.

[72] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[73] P. Van Hentenryck. *Consistency Techniques in Logic Programming.* PhD thesis, University of Namur, Belgium, 1987.

[74] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.

[75] P. Vasey. Qualified answers and their application to transformation. In *Proceedings of the Third International Logic Programming Conference*, pages 425–432, 1986.

[76] P. Voda. The constraint language Trilogy: Semantics and computations. Technical report, Complete Logic Systems, North Vancouver, B.C., Canada, 1988.

[77] J.E. Vuillemin. Exact real computer arithmetic with continued fractions. *IEEE Transactions on Computers*, 37(8):1087–1105, August 1990.

[78] D. Waltz. Understanding line drawings of scenes with shadows. In P. Winston, editor, *The Psychology of Computer Vision*. McGraw-Hill, 1975.

[79] C.M. White. *ALS Prolog User's Guide and Reference Manual—Sun Workstation Version 1.01*. Applied Logic Systems, Inc., August 1988.

[80] M. Zweben and M. Eskey. Constraint satisfaction with delayed evaluation. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 875–880, Detroit, USA, 1989.

Appendix A

Consistency Techniques In Logic Programming

This appendix reviews the concepts of consistency techniques in logic programming required for discussion in the thesis. We refer the readers to [73, 74] for detailed treatment of the subject.

A.1 Domain Variables

A *domain* is a non-empty finite set of constants. Let $\mathcal{D}$ be the set of all domains we possibly need during the computation. $\mathcal{D}$ has the property that $D \in \mathcal{D}$ implies $E \in \mathcal{D}$ for all non-empty subsets E of D. To extend a first-order language to include domain variables, we need to add $\mathcal{D}$ to the language and also a set $\mathcal{V}$ of domain variables with domains from $\mathcal{D}$. The usual variables of first-order logic are called *simple variables*. *Domain variables* with domain D are superscripted with D, as in X^D. Intuitively, a domain variable X^D can be bound only to elements of D. Terms, atoms, literals,

definite clauses and programs can be defined as usual [45] except that variables can either be simple variables or domain variables. The meaning of the $\forall$ and $\exists$ quantifiers can be extended in a straight-forward manner to accommodate domain variables and the notions of logical consequence, Herbrand interpretation, Herbrand model as well as the least Herbrand model can be defined as usual [73].

Definition A.1.1

A ***domain substitution*** θ is a finite set of the form $\{V_1/t_1, \ldots, V_n/t_n\}$, where

1. each V_i is a variable;
2. each t_i is a term distinct from V_i;
3. all the V_i's are distinct;
4. if V_i is a domain variable X^D, then t_i is a constant included in D or is a domain variable Y^E with $E \subseteq D$.

Each element V_i/t_i is called a binding for V_i. θ is called a *ground domain substitution* if each t_i is a ground term for $i = 1, \ldots, n$. An *expression* is either a term, a literal, or a conjunction of literals. Let $\theta = \{V_1/t_1, \ldots, V_n/t_n\}$ be a domain substitution and K be an expression. Then $K\theta$, the *instance* of K by θ, is the expression obtained from K by simultaneously replacing each occurrence of the variable V_i in K by the term t_i $(1 \leq i \leq n)$. This definition of domain substitution makes sure that domain variables are only instantiated with elements from their respective domains.

The composition of domain substitutions is defined as that of substitutions in [45]. Unifier is defined in the usual way, but the definition of most general unifier (mgu) is a generalization of the standard terminology. An extended unification algorithm that handles domain variables is presented in [74]. The three following cases should be added to the usual algorithm:

1. If a domain variable and a constant have to be unified, the unification succeeds if the constant is in the domain of the domain variable and binds the latter to the former. Otherwise the unification fails.

2. If two domain variables have to be unified, the unification succeeds if the intersection of their domains is non-empty and binds both variables to a new domain variable ranging over this intersection. Otherwise the unification fails.

3. If a domain variable and a simple variable have to be unified, the unification succeeds and binds the simple variable to the domain variable.

SLD-resolution with this extended unification algorithm is called SLDD-resolution. The proofs of the soundness and completeness of SLDD-resolution can be based on those of SLD-resolution [45], which do not take into account the fact that unifications take place in an empty and unsorted equational theory.

A.2 Consistency Techniques

The proof procedure for logic programming with domains is based on SLDD-resolution and three new inference rules based on consistency techniques [46]: forward checking, looking ahead, and partial looking ahead. Looking ahead is a generalization of forward checking. Partial looking ahead, which is used to justify the implementation of some built-in predicates, is an approximation of looking ahead. For this thesis, it is sufficient to restrict our discussion to the *Looking-Ahead Inference Rule* (LAIR).

We begin by defining the kind of predicates, to which LAIR can be applied. An n-ary predicate p is a *domain constraint* if and only if for any ground terms $g_1, \ldots, g_n$, either $p(g_1, \ldots, g_n)$ has a successful refutation or $p(g_1, \ldots, g_n)$ has only finitely failed

derivations. In general, it is a semi-decidable problem to decide whether a predicate is a domain constraint. In the CHIP language [22], it is the programmer's responsibility to declare a predicate to be a domain constraint. A domain constraint $p(t_1, \ldots, t_n)$ is *lookahead-checkable* if and only if there exists at least one t_i that is a domain variable, and each of the other arguments is either ground or a domain variable. The domain variables in $t_1, \ldots, t_n$ are called the *lookahead variables*. We have now enough concepts to define LAIR.

Definition A.2.1 (LAIR)
Let P be a program and $G_i = \leftarrow A_1, \ldots, A_m, \ldots, A_k$ be a goal. G_{i+1} is *derived by LAIR* from G_i and P using the domain substitution θ_{i+1} if the following conditions hold:

1. A_m is lookahead-checkable and $X_1, \ldots, X_n$ are the lookahead variables of A_m, which range respectively over $D_1, \ldots, D_n$.

2. For all $1 \leq j \leq n$, $E_j = \{a_j \in D_j \,|\, \exists a_1 \in D_1, \ldots, a_{j-1} \in D_{j-1}, a_{j+1} \in D_{j+1}, \ldots, a_n \in D_n$ such that $P \models A_m\theta\} \neq \emptyset$, where $\theta = \{X_1/a_1, \ldots, X_n/a_n\}$.

3. z_i is the constant c if $E_j = \{c\}$ or a new domiain variable ranging over E_j otherwise.

4. θ_{i+1} is $\{X_1/z_1, \ldots, X_n/z_n\}$.

5. G_{i+1} is either $\leftarrow (A_1, \ldots, A_{m-1}, A_{m+1}, \ldots, A_k)\theta_{i+1}$ if at most one z_i is a domain variable, or $\leftarrow (A_1, \ldots, A_k)\theta_{i+1}$ otherwise.

A soundness result for the LAIR is presented in [73] but the completeness result does not hold in general. However, a sound and complete proof procedure using the LAIR and SLDD-derivation can still be programmed but it may be necessary to use normal derivation for lookahead-checkable predicates.

A.3 Using LAIR In CHIP

In CHIP, the programmer is responsible for specifying the inference rule to be used for atoms with particular predicate names and also the preconditions for its use. A ***lookahead declaration*** for an n-ary predicate p is a unique expression for p of the following form:

lookahead $p(a_1, \ldots, a_n)$,

where each a_i is either 'g' or 'd'. All atoms with predicate p are said to be ***submitted*** to this lookahead declaration. Atoms submitted to lookahead declarations are reduced by LAIR during derivations. A lookahead declaration also specifies the preconditions that an atom has to satisfy before it can be reduced. An atom A submitted to a lookahead declaration is *lookahead-available* if and only if

1. all the arguments of A corresponding to a 'g' in the lookahead declaration are ground;

2. A is lookahead-checkable.

The lookahead-efficient computation rule used in CHIP only selects an atom, submitted to a lookahead declaration, after it becomes lookahead-available.

Appendix B

The CLP Scheme And CLP($\Re$)

This appendix briefly reviews the CLP scheme and CLP($\Re$). We refer the readers to [68, 32] for detailed treatment of the subject. We adopt the standard terminology from [45].

The CLP scheme [33] generalizes logic programming by replacing unification with constraint solving. The scheme defines a family of constraint logic programming languages CLP($\mathcal{X}$), parameterized by a domain of computation $\mathcal{X}$. These languages share the same semantic properties. Constraint solving is not new in computing. What is new is that it is embraced in the framework of logic programming, which has a simple syntax and declarative semantics.

Associated with every constraint domain $\mathcal{X}$ are a model $\mathcal{M}_\mathcal{X}$ and a theory $\mathcal{T}_\mathcal{X}$ that axiomatizes $\mathcal{M}_\mathcal{X}$. Jaffar and Lassez [32, 33] show that if $\mathcal{M}_\mathcal{X}$ is solution-compact (page 100) and $\mathcal{T}_\mathcal{X}$ is satisfaction-complete (page 102), then all the basic results in traditional logic programming [72, 4] hold. Both $\mathcal{M}_\mathcal{X}$ and $\mathcal{T}_\mathcal{X}$ are defined with respect to a many-sorted first-order language that consists of a collection $\Sigma_\mathcal{X}$ of function symbols (constants are functions of zero arity) and a collection $\Pi_\mathcal{X}$ of predicate symbols.

A *sort* is a non-empty set. We use SORT = $\bigcup(\text{SORT}_i)$ to denote the finite set of sorts. A *signature* of an n-ary function (predicate and variable) symbol f is a sequence of $n+1$ (respectively n and 1) elements of SORT. By the term *sort of* f, we mean the last element in the signature of the function symbol f. While the symbol $\mathcal{V}$ denotes a collection of variables, we use $\tau(\Sigma_{\mathcal{X}})$ and $\tau(\Sigma_{\mathcal{X}} \cup \mathcal{V})$ to denote the set of well-typed ground terms and terms that may contain variables respectively.

B.1 Structure

A structure $\mathcal{M}_{\mathcal{X}}$ consists of

1. a collection $D\mathcal{X}$ of non-empty sets $D\mathcal{X}_s$, where s ranges over the sorts in SORT,
2. an assignment, to each n-ary $f \in \Sigma_{\mathcal{X}}$, of a function

   $$D\mathcal{X}_{s_1} \times D\mathcal{X}_{s_2} \times \cdots D\mathcal{X}_{s_n} \rightarrow D\mathcal{X}_s$$

   where $(s_1, s_2, \ldots, s_n, s)$ is the signature of f, and finally
3. an assignment, to each n-ary $p \in \Pi_{\mathcal{X}}$, of a function

   $$D\mathcal{X}_{s_1} \times D\mathcal{X}_{s_2} \times \cdots D\mathcal{X}_{s_n} \rightarrow \{\text{true}, \text{false}\}$$

   where $(s_1, s_2, \ldots, s_n)$ is the signature of p.

An *atomic constraint* is of the form $p(t_1, \ldots, t_m)$, where $p \in \Pi_{\mathcal{X}}$ and $t_i \in \tau(\Sigma_{\mathcal{X}} \cup \mathcal{V})$ for $i = 1, \ldots, m$. A *constraint* is a finite set, possibly empty, of atomic constraints. A constraint is interpreted as an existentially closed conjunction of positive atomic constraints.

For a structure to be usable in a computing framework, we require that the syntax can distinguish and specify all elements in the structure. To restrict ourselves to structures with this property, we impose the condition of "solution compactness."

A many-sorted structure $\mathcal{M}_\mathcal{X}$ is *solution-compact* if

1. every element in $\mathcal{M}_\mathcal{X}$ is the unique solution of a finite or infinite set of constraints; and
2. every element in the complement of the solution space of a constraint c, belongs to a disjoint union of solution space of some finite or infinite family of constraints c_i.

The elements, which can only be defined as the unique solution of an infinite constraint set, are called *limit elements*. Symbolically, we can write the conditions as

1. $\forall d \in \mathcal{M}_\mathcal{X} \exists c_i : d = \bigcap c_i$,
2. $\forall c \, \exists c_i : \overline{c} = \bigcup c_i$.

An $\mathcal{M}_\mathcal{X}$ *valuation* on an expression over $\Pi_\mathcal{X}$, $\Sigma_\mathcal{X}$, and $\mathcal{V}$ is a mapping from each distinct variable in the expression into $D\mathcal{X}_s$, where s is the sort of the variable. Let θ be an $\mathcal{M}_\mathcal{X}$ valuation and t a term. We write $t\theta$ to denote the application of θ to t; thus $t\theta$ is an element in $D\mathcal{X}_s$, where s is the sort of t. Similarly, for an atomic constraint c, $c\theta$ denotes the application of θ to c. We denote the truth and falsity of $c\theta$ in the model $\mathcal{M}_\mathcal{X}$ respectively by $\models_{\mathcal{M}_\mathcal{X}} c\theta$ and $\models_{\mathcal{M}_\mathcal{X}} \neg c\theta$. If C is a set, possibly infinite, of atomic constraints, we write $\models_{\mathcal{M}_\mathcal{X}} C\theta$ if $\models_{\mathcal{M}_\mathcal{X}} c\theta$ for all $c \in C$ and, $\models_{\mathcal{M}_\mathcal{X}} \neg C\theta$ otherwise. If there is a θ such that $\models_{\mathcal{M}_\mathcal{X}} C\theta$, we say that C is *solvable in* $\mathcal{M}_\mathcal{X}$ and that θ is an $\mathcal{M}_\mathcal{X}$ *solution* of C.

B.2 $\mathcal{M}_{\mathcal{X}}$ Models

A *CLP($\mathcal{X}$) constraint logic program* is defined over disjoint (sorted) alphabets Π_P and $\Sigma_{\mathcal{X}}$. An *atom* is of the form $p(t_1, \ldots, t_n)$, where $p \in \Pi_P$ and $t_i \in \tau(\Sigma_{\mathcal{X}} \cup \mathcal{V})$ for $i = 1, \ldots, n$. A CLP($\mathcal{X}$) program P consists of a finite set of clauses (rules) of the form

$$A \leftarrow c, B_1, \ldots, B_n$$

where $n \geq 0$, A and B_i are atoms for $i = 1, \ldots, n$, and c is a constraint. A is called the *head* of the clause and $c, B_1, \ldots, B_n$ is the *body*. A *goal* is a clause without a head:

$$\leftarrow c, B_1, \ldots, B_n.$$

The $\mathcal{M}_{\mathcal{X}}$ *base* of a program P is

$$\{p(t_1, \ldots, t_n) \,|\, p \in \Pi_P \text{ and } t_i \in \tau(\Sigma_{\mathcal{X}}) \text{ for } i = 1, \ldots, n\}.$$

An $\mathcal{M}_{\mathcal{X}}$ *interpretation* I_P of P is a subset of the $\mathcal{M}_{\mathcal{X}}$ base of P. A clause in P

$$A \leftarrow c, B_1, \ldots, B_n, \qquad \text{where } n \geq 0,$$

is *true* in I_P if, for every $\mathcal{M}_{\mathcal{X}}$ valuation θ on A, c, and B_i ($i = 1, \ldots, n$) such that θ is an $\mathcal{M}_{\mathcal{X}}$ solution of c,

$$\{B_1\theta, \ldots, B_n\theta\} \subseteq I_P \Rightarrow A\theta \in I_P.$$

An $\mathcal{M}_{\mathcal{X}}$ *model* of P is an $\mathcal{M}_{\mathcal{X}}$ interpretation I_P of P such that every rule in P is true in I_P. An $\mathcal{M}_{\mathcal{X}}$ model is a model in the usual model-theoretic sense. Jaffar and

Lassez [32, 33] show the existence of the least and greatest $\mathcal{M}_\mathcal{X}$ models. Let A be a conjunction of ground atoms. A is a *logical consequence* of P, written $P \models_{\mathcal{M}_\mathcal{X}} A$, if and only if each atom in A is an element of the least $\mathcal{M}_\mathcal{X}$ model of P. Given a constraint c and a conjunction of atoms A, we write

$$P \models_{\mathcal{M}_\mathcal{X}} \exists(c, A),$$

to mean that

1. $\models_{\mathcal{M}_\mathcal{X}} \exists(c)$, and
2. if θ is an $\mathcal{M}_\mathcal{X}$ solution of c, then $P \models_{\mathcal{M}_\mathcal{X}} \exists(A\theta)$.

B.3 Logical Theory

Let $\mathcal{T}_\mathcal{X}$ be a first-order theory that axiomatizes $\mathcal{M}_\mathcal{X}$. Well-formed formulae, truth, model, satisfiability, etc, are defined in the usual way, such as in [45]. To obtain negative information from a theory $\mathcal{T}_\mathcal{X}$, we require a further condition. A theory $\mathcal{T}_\mathcal{X}$ is *satisfaction-complete* if

$$\mathcal{T}_\mathcal{X} \models \forall(\neg c) \text{ whenever not } \mathcal{T}_\mathcal{X} \models \exists(c),$$

for all constraints c, where $\forall(c)$ and $\exists(c)$ denote the universal and existential closures of c respectively. Given a CLP($\mathcal{X}$) program P, we are interested in the logical consequences of $P \cup \mathcal{T}_\mathcal{X}$. In principle, we want the theory to be as close to the structure as possible. $\mathcal{M}_\mathcal{X}$ and $\mathcal{T}_\mathcal{X}$ *correspond* if

1. $\models_{\mathcal{M}_\mathcal{X}} \mathcal{T}_\mathcal{X}$ ($\mathcal{M}_\mathcal{X}$ is a model of $\mathcal{T}_\mathcal{X}$), and

2. if $\models_{\mathcal{M}_\mathcal{X}} \exists(c)$, then $\mathcal{T}_\mathcal{X} \models \exists(c)$, for all constraints c, where $\exists(c)$ denotes the existential closure of c.

In other words, the truth in $\mathcal{M}_\mathcal{X}$ has a one-one correspondence to satisfiability in $\mathcal{T}_\mathcal{X}$. This allows us to study the semantics of programs in either framework. We shall focus on the model-theoretic viewpoint in the discussion of the operational semantics of the CLP scheme.

B.4 Operational Semantics

Let P be a CLP($\mathcal{X}$) program and $G_i = \leftarrow c, B_1, \ldots, B_n$, $n > 0$, be a goal. G_{i+1} is *derived in* $\mathcal{M}_\mathcal{X}$ from G_i if

1. there exists in P a collection of n variants of clauses

   $$A_1 \leftarrow c_1, \vec{B}_1$$
   $$A_2 \leftarrow c_2, \vec{B}_2$$
   $$\ldots$$
   $$A_n \leftarrow c_n, \vec{B}_n$$

   where $\vec{B}_j$ is a set of atoms for $j = 1, \ldots, n$, such that

   $$c' = c \cup c_1 \cup \cdots \cup c_n \cup \{B_1 = A_1, \ldots, B_n = A_n\}$$

   is solvable in $\mathcal{M}_\mathcal{X}$, and

2. $G_{i+1} = \leftarrow c', \vec{B}_1, \ldots, \vec{B}_n$.

An $\mathcal{M}_\mathcal{X}$ *derivation* is a, possibly infinite, sequence of goals $G = G_0, G_1, G_2, \ldots$ such that G_{i+1} is derived in $\mathcal{M}_\mathcal{X}$ from G_i. An $\mathcal{M}_\mathcal{X}$ derivation is *successful* if it is finite

and the last goal contains no atoms. In this case, the constraint in the last goal is called the ***answer constraint***. An $\mathcal{M}_{\mathcal{X}}$ derivation is ***finitely-failed*** if it is finite and the last goal has one or more atoms and condition 1 does not hold.

$\mathcal{T}_{\mathcal{X}}$ derivations, derivations in the logical theory $\mathcal{T}_{\mathcal{X}}$, can be defined similarly except that the test of solvability in $\mathcal{M}_{\mathcal{X}}$ in a derivation step is replaced by a test of satisfiability in the theory $\mathcal{T}_{\mathcal{X}}$. Jaffar and Lassez [32, 33] show the soundness and completeness of $\mathcal{M}_{\mathcal{X}}$ and $\mathcal{T}_{\mathcal{X}}$ derivations.

B.5 CLP(ℜ)

CLP(ℜ) is an instance of the CLP scheme, where ℜ is the domain of finite trees over real numbers. In the structure $\mathcal{M}_{\Re}$, we consider two sorts, s_1 and s_2. The sort s_1 is the set of real numbers and s_2 is the set of finite trees of real numbers. Thus, $s_1 \subseteq s_2$. $\Sigma_{\Re}$ contains the arithmetic function symbols $\{+, \times, 0, 1, 2, \ldots\}$ of sort s_1 and uninterpreted symbols $\{+, \times, 0, 1, 2, \ldots, f, g, \ldots\}$ of sort s_2. The functional interpretation for the first set is the usual one; for the second set, an n-ary f maps a sequence of elements in s_2 into the tree whose root node is labeled with f and whose children are given by its arguments. $\Pi_{\Re}$ is $\{=, <, \leq\}$. The signature of "$=$" is (s_2, s_2); while those of "$<$" and "$\leq$" are both (s_1, s_1). The relations among real numbers are defined in the usual way. Two trees, T_1 and T_2, are equal if

1. T_1 and T_2 are real number terms and $T_1 = T_2$, or
2. T_1 or T_2 are not real number terms and their root functors are the same and their corresponding subtrees, if any, are the same.

The corresponding theory $\mathcal{T}_{\Re}$ of ℜ is the theory RCF of the real closed field [70, 16]. $\mathcal{M}_{\Re}$ is solution-compact and RCF is satisfaction-complete [32].