A Generic, Collaborative Framework for Interval Constraint Solving

PhD Thesis

Departamento de Lenguajes y Ciencias de la Computación
University of Málaga, Spain

By

Antonio J. Fernández Leiva

Director: Dr. Patricia M. Hill
School of Computing
University of Leeds, England

Tutor: Dr. José M. Troya
Departamento de Lenguajes y Ciencias de la Computación
University of Málaga, Spain

January, 25th 2002

Acronyms

Acronym	*Meaning*
AI	Artificial Intelligence
ATOAM	yet Another matching Tree Oriented Abstract Machine
CCL	Concurrent Constraint Logic
CCP	Concurrent Constraint Programming
CHR	Constraint Handling Rule
CLP	Constraint Logic Programming
CLP(B)	Constraint Logic Programming on the Boolean domain
CLP(FD)	Constraint Logic Programming on the finite domain
CLP($\Re$)	Constraint Logic Programming on the real domain
CP	Constraint Programming
CSP	Constraint Satisfaction Problem
FD	Finite Domain
LP	Logic Programming
OR	Operational Research
RC	Reified Constraint
SRQ	Self Referential Quiz
WAM	Warren Abstract Machine
2D	2-Dimensional

Abstract

This thesis deals with a generic and cooperative schema for CLP(Interval($\mathcal{X}$)) where $\mathcal{X}$ is any computation domain with lattice structure. This schema, based on interval lattices, is a general framework for interval constraint satisfaction and interval solver cooperation on domains with lattice structure independently of its cardinality. Our proposal assures a complete glass box setting on which both constraints and domains as well as the intended propagation and cooperation mechanisms among constrained variables can be easily defined from the user level. The main body of the thesis presents a formal specification of this schema.

This thesis presents the following main results:

- A comprehensive comparison of both the efficiency and certain aspects of the expressiveness of a number of constraint systems. This comparison, done over the Boolean and the finite domains, illustrates main differences between existing constraint systems.

- We propose a constraint satisfaction framework for CLP(Interval($\mathcal{X}$)) by describing the whole process of interval constraint solving on any domain with lattice structure, detailing separately the processes of interval propagation and interval branching. One of the advantage of our proposal is that monotonicity of constraints is implicitly defined in the theory. Also, we present a statement of a number of interesting properties that, subject to certain conditions, are satisfied by any instances of the schema. Moreover, we show that many constraint systems satisfy these conditions and point out other non trivial and interesting instances of our framework.

- Our schema for CLP(Interval($\mathcal{X}$)) enables solver cooperation by allowing the information flow between distinct computation domains. This gives rises to the possibility of a mix of different instances of the schema e.g., well known instances such as CLP(Interval($\Re$)), CLP(Interval(*Integer*)), CLP(Interval(*Set*)), CLP(Interval(*Bool*)) and new instances resulting from user defined domains or even from the combination of existing domains in the way CLP(Interval($\mathcal{X}_1 \times \ldots \times \mathcal{X}_n$)). Therefore $\mathcal{X}$ may be instantiated to a set of lattice structure computation domains and the corresponding CLP(Interval($\mathcal{X}$)) allows multiple flexibility in the definition of (probably user-defined) domains in $\mathcal{X}$ and interaction between them.

- By means of a prototype implementation we show that a single system based on our CLP(Interval($\mathcal{X}$)) schema may provide support for classical interval constraint satisfaction and optimisation as well as for interval solver cooperation over a multiple set of computations domains. Moreover the system is a glass box approach from a double perspective since the user can define not only new constraints and the intended propagation mechanism but also new domains on which constraints can be solved and the intended cooperation mechanism between all the (user or system defined) computation domains.

  In our opinion, this opens up new potential directions to research inside the interval constraint community.

To reach the aims, this thesis has followed the following steps (1) the election of an adequate approach to support the generic schema (2) the construction of a generic theoretical framework (called *the basic framework*) to propagate interval constraints on any domain with lattice structure, (3) the integration in the basic framework of a novel technique of solver cooperation by means of constraint operators and (4) the extension of the resulting setting to solve and optimise interval constraints.

(1) There are two key reasons for adopting CLP technology for solving a problem. The first is its expressiveness enabling a declarative solution with readable code which is vital for maintenance and the second is the provision of an efficient implementation for the computationally expensive procedures. However, CLP systems differ significantly both in how solutions may be expressed and the efficiency of their execution and it is important that both these factors are taken into account when choosing the best CLP system for a particular application. Also among the domains of CLP, the FD is one of the most studied since a lot of problems involve variables ranging in discrete domains.

  Currently, there are several techniques to support constraint solving on FD in the CLP systems. We have developed a comparison on the efficiency of a number of CLP systems in the setting of finite domains as well as a specific aspect of their expressiveness (that concerning reification and meta-constraints). We have compared eight systems that, strictly speaking, are glass boxes since they allow user defined constraints in a more or less clear way. This comparison illustrates differences between the systems, indicating their particular strengths and weaknesses and helps in the choice of the best technique for building our generalised framework for interval constraint solving.

(2) From the comparison done previously, we choose the transparent approach called *indexical* due to its flexibility, its simplicity and its performance. We generalise the indexical approach on FD for interval constraint propagation to domains with lattice structure. We provide the theoretical foundations for this framework, a schematic procedure for the operational semantics, and numerous examples illustrating how it can be used both over classical and new domains. We also show how lattice combinators can be used to generate new domains and hence

new constraint solvers for these domains from existing domains. As most of the existing domains are lattices, our framework provides support for them.

(3) In our CLP(Interval($\mathcal{X}$)) schema for constraint propagation devised previously, the interval constraint solvers are each based on the same generic solver so that they are completely independent from each other and there is no provision for any cooperation between them. Therefore, we extend the theoretical basic generic framework to enable solver cooperation and allow information to flow between different computation domains. This is done by means of a novel technique allowing constraint operators to be defined over multiple domains enabling thus a one-way communication channel between different domains. To allow for a two way channel we define the generic concept of *high level constraint*. As consequence the different solvers can communicate and hence, cooperate in solving the problem.

Steps (2) and (3) have been integrated in this thesis.

(4) Often, constraint propagation is not enough to solve completely a CSP and some additional strategy must be applied for it. Therefore, we have completed the cooperative CLP(Interval($\mathcal{X}$)) schema for constraint propagation by proposing a parameterised CLP(Interval($\mathcal{X}$)) schema for constraint branching that (with suitable instantiations of the parameters) can solve completely CSPs defined via interval constraints on any set of domains $\mathcal{X}$ with lattice structure.

Finally we introduce *clp*($\mathcal{L}$), an interval constraint logic programming language that allows constraint solving on any set $\mathcal{L}$ of lattices that is based on our proposal. We also give an outline of a prototype implementation and some examples of use. This prototype implementation demonstrates that a single system based on our CLP(Interval($\mathcal{X}$)) schema is enough to provide support for multiple domains, solver cooperation, solver satisfaction and solver optimisation in a glass box setting on both constraints and domains.

Resumen (in spanish)

Esta tesis propone un esquema genérico y cooperativo para CLP(Interval($\mathcal{X}$)) donde $\mathcal{X}$ es cualquier dominio de computación con estructura de retículo. El esquema, que está basado en la teoría de retículos, es un enfoque general para la satisfacción y optimización de restricciones de intervalo así como para la cooperación de resolutores de intervalo definidos sobre dominios de computación con estructura de retículos, independientemente de la cardinalidad de éstos. Nuestra propuesta asegura un enfoque transparente sobre el cual las restricciones, los dominios de computación y los mecanismos de propagación y cooperación, definidos entre las variables restringidas, pueden ser fácilmente especificados a nivel del usuario. La parte principal de la tesis presenta una especificación formal de este esquema.

Los principales resultados conseguidos en esta tesis son los siguientes:

- Una comparativa global de la eficiencia y algunos aspectos de la expresividad de ocho sistemas de restricciones. Esta comparativa, realizada sobre el dominio finito y el dominio Booleano, muestra diferencias principales entre los sistemas de restricciones existentes.

- Para formalizar el marco de satisfacción de restricciones para CLP(Interval($\mathcal{X}$)) hemos descrito el proceso global de resolución de restricciones de intervalo sobre cualquier retículo, separando claramente los procesos de propagación y división (ramificación) de intervalos. Una de las ventajas de nuestra propuesta es que la monotonía de las restricciones está implícitamente definida en la teoría. Además, declaramos un conjunto de propiedades interesantes que, bajo ciertas condiciones, son satisfechas por cualquier instancia del esquema genérico. Más aún, mostramos que muchos sistemas de restricciones actualmente existentes satisfacen estas condiciones y, además, proporcionamos indicaciones sobre cómo extender el sistema mediante la especificación de otras instancias interesantes y novedosas.

- Nuestro esquema para CLP(Interval($\mathcal{X}$)) permite la cooperación de resolutores de manera que la información puede fluir entre diferentes dominios de computación. Además, es posible combinar distintas instancias del esquema: por ejemplo, instancias bien conocidas tales como CLP(Interval($\Re$)), CLP(Interval(*Integer*)), CLP(Interval(*Set*)), CLP(Interval(*Bool*)), y otras novedosas que son el resultado de la generación de nuevos dominios de computación definidos por el usuario, o

incluso que surgen de la combinación de dominios ya existentes como puede ser CLP(Interval($\mathcal{X}_1 \times \ldots \times \mathcal{X}_n$)). Por lo tanto, $\mathcal{X}$ puede ser instanciado a cualquier conjunto de dominios de computación con estructura de retículo de forma que su correspondiente instancia CLP(Interval($\mathcal{X}$)) permite una amplia flexibilidad en la definición de dominios en $\mathcal{X}$ (probablemente definidos por el usuario) y en la interacción entre estos dominios.

- Mediante la implementación de un prototipo, demostramos que un único sistema, que esté basado en nuestro esquema para CLP(Interval($\mathcal{X}$)), puede proporcionar soporte para la satisfacción y la optimización de restricciones así como para la cooperación de resolutores sobre un conjunto conteniendo múltiples dominios de computación. Además, el sistema sigue un novedoso enfoque transparente sujeto a una doble perspectiva ya que el usuario puede definir no sólo nuevas restricciones y su mecanismo de propagación, sino también nuevos dominios sobre los cuales nuevas restricciones pueden ser resueltas así como el mecanismo de cooperación entre todos los dominios de computación (ya sean definidos por el usuario o predefinidos por el sistema).

  En nuestra opinión, esta tesis apunta nuevas líneas de investigación dentro de la comunidad de las restricciones de intervalo.

Para alcanzar los resultados expuestos, hemos seguido los siguientes pasos (1) la elección de un enfoque adecuado sobre el cual construir los fundamentos teóricos de nuestro esquema genérico; (2) la construcción de un marco teórico genérico (que llamaremos el *marco básico*) para la propagación de restricciones de intervalo sobre cualquier retículo; (3) la integración, en el marco básico, de una técnica novedosa que facilita la cooperación de resolutores y que surge de la definición, sobre múltiples dominios, de operadores de restricciones y (4) la extensión del marco resultante para la resolución y optimización completa de las restricciones de intervalo.

Finalmente presentamos *clp*($\mathcal{L}$), un lenguaje de programación lógica de restricciones de intervalo que posibilita la resolución de restricciones sobre cualquier conjunto de retículos y que está implementado a partir de las ideas formalizadas en el marco teórico. Describimos una primera implementación de este lenguaje y desarrollamos algunos ejemplos de cómo usarla. Este prototipo demuestra que nuestro esquema para CLP(Interval($\mathcal{X}$)) puede ser implementado en un sistema único que, como consecuencia, proporciona, bajo un enfoque transparente sobre dominios y restricciones, cooperación de resolutores así como satisfacción y optimización completa de restricciones sobre diferentes dominios de computación.

Nota del autor. Existe una versión simplificada de la tesis en español. En esta versión, todas las demostraciones de teoremas, lemas y proposiciones han sido omitidas, puesto que el objetivo de la misma es aportar un resumen (extendido) de la tesis completa. La versión en español está disponible en la siguiente dirección:

`http://www.lcc.uma.es/~afdez/Papers/phdspa3nish.pdf`

Acknowledgements

Overall, I thank Pat (my director) for the improvement of the ideas applied in this thesis, making them coherent and presentable at the same time. And the most important of all: in spite of being a very busy person, she always was there when I really needed it. For all this, my major thanks and my eternal friendship.

I am grateful to José María Troya for providing me with the opportunity to become a post-graduate research fellow. Probably, without this fellowship, I would not be working here at the University of Málaga. I also thank him for his trust in me and his encouragement for me to visit Leeds and work with Pat.

Thanks to Ernesto Pimentel and Juan Miguel Molina for a general look at the contents of this thesis and for reading and commenting on some of its chapters. They helped improve the presentation of this document. Also Ernesto helped me in the administrative steps to present this thesis. I thank him for his time.

I want to mention the many wonderful friends I made in Leeds that contributed to the happy times there. Among them I want to thank especially Fausto Spoto, Karim Kjemame, Javier Nuñez and again Pat.

I would also like to mention specially a colleague and friend of my department, Pablo López. The daily conversations with him, usually more concerned with life research than with science research, have contributed to marvelous hours spent in my University (particularly when enjoying lunch together).

During the first years of my research, I was supported by a FPI grant of the MEC (Ministerio de Educación y Ciencia de España). During the last two years I have been supported by CICYT grant TIC98-0445-C03-03. My thanks to MEC and CICYT for this.

As well, I thank my family (wife, parents and brothers) for their continuing support. I know they are proud of me. I am also proud of them.

I am dedicating this thesis to my beloved son Angel. He has shown me what infinite love means.

Preface

This thesis is composed of 4 parts that we try to make relatively independent.

Part I is composed of two chapters. Chapter 1 motivates the generic, cooperative and transparent system for interval constraint solving described in this thesis. This is done by discussing the limitations of the current instances of CLP and by showing how these are solved in our proposal. Chapters 2 provides a general overview of the basics over which CLP is founded.

Part II describes a comprehensive comparison over the Boolean and FD of different constraint systems subjected to different approaches and distinct mechanisms of constraint solving. The comparison is done not only from the performance point of view but also from specific aspects of the expressiveness. This part is developed in Chapter 3.

Part III is dedicated to the formalisation of the generic theoretical framework that we propose for interval constraint solving as well as interval solver cooperation on any set of computation domains with lattice structure. This part begins by proposing a generic schema for interval constraint propagation on lattices that is developed in Chapter 4. Then, Chapter 5 extends this framework to enable solver cooperation. Finally in Chapter 6 we complete the theoretical framework by describing a generic schema for branching of interval constraints. We show that this schema allows classical constraint solving as well as constraint optimisation. This part is also dedicated to study a number of interested properties of the schema, to develop numerous examples to show the declarativity of the generic setting and to treat other issues such as constraint monotonicity, high level constraints and combination of domains.

Part IV is devoted to describe a prototype implementation of our theoretical framework inside a logic programming language. A number of non-standard examples are described to show the declarativity and flexibility of the resulting system. This part is developed in Chapter 7.

The thesis terminates with a chapter that briefly summarises the results and gives major directions for future works and improvements.

Presentations

The original motivations of our work described in Chapter 1 of Part I were initially presented in (Fernández, 1997).

Part II was almost integrally published in the *Constraints* journal (Fernández and

Hill, 2000a). Previously, the Sections about SRQs (i.e., Sections 3.3, 3.4 and 3.5.2) were presented in (Fernández and Hill, 1997a) and (Fernández and Hill, 1997b). A summary of Sections 3.5.3 and 3.5.4 were also published in (Fernández and Hill, 1998c).

With respect to Part III, a preliminary version of Chapter 4 was published in (Fernández and Hill, 1999c). Previous versions of this work were presented in (Fernández and Hill, 1998b; Fernández and Hill, 1998a) and (Fernández and Hill, 1999a). Chapters 5 and 6 were presented in (Fernández and Hill, 2000b) and (Fernández and Hill, 2001a) respectively. Also a journal version of Chapters 4 and 5 is now being revised (Fernández and Hill, 2001b).

Part IV is available, as user manual, in (Fernández, 2000).

Contents

List of Figures

List of Tables

Part I

Introduction and Background

Chapter 1

Introduction and Motivation

It isn't that they can't see the solution.
It is that they can't see the problem.

The Scandal of Father Brown (1935)
G.K. Chesterton, 1874-1936

1.1 Introduction and Motivation

Constraint Logic Programming (CLP) (Jaffar and Maher, 1994) was born from a desire to solve problems that could not be solved by logic programming (LP) alone. Its success lies in that it combines the declarativity of LP with the efficiency of the constraint programming (CP) paradigm. However, although there has been considerable research in the area, CLP is a relatively new field in which current systems still have many evident limitations.

In this chapter, we discuss some of these limitations and outline possible solutions for each of them. To provide a suitable framework for these solutions in a single setting is the main objective of this thesis which, in summary, is to find a generic, cooperative, transparent and efficient CLP system.

A generic schema for CLP. The essential component of the CLP schema is that it can be parameterised by a computation domain in such a way that different types of the domain determine different instances of the schema; for instance, we have CLP(FD) (CLP on finite ranges of integers), CLP($\Re$) (CLP on real domain), CLP(Sets) (CLP on finite sets of elements) and CLP(Bool) (CLP on the Boolean domain). CLP systems have been applied to many different domains and each CLP instance may be identified by the mechanism used for constraint solving and by its algebraic structure. It is well known that the expressive power and efficiency of CLP systems is reduced by the strong partitioning of the structures (i.e., the objects -values in the computation

Figure 1.1: Specific reasoning for distinct domains

domain- and the operations on them) in which the constraints may be expressed. This means that constraints must be defined on a specific CLP domain and solved by the reasoning method and associated solver associated with that domain (see Figure 1.1). In particular, with existing CLP systems, the cardinality of a domain determines the form of the constraint solving procedure and there are quite distinct approaches to constraint solving for the finite and the infinite domains.

The advantage of a domain-specific solver is that it can be optimised for the particular characteristics of the algebraic structure of the domain, leading to a better performance of the solver. However, a domain-specific solver has obvious disadvantages.

- First, as already pointed out, a domain-specific solver is not able to solve con-

straints on other computation domains and its application context is merely restricted to the domain for which was designed. This is an important limitation since in practice, problems themselves are not specific to any particular domain and thus the formulation of a problem has to be artificially adapted to fit a given solver, probably losing, as a consequence, part of the declarativity of the solution.

- Secondly, despite the fact that its (declarative and operational) semantics may be projected from the CLP schema, there remain a number of properties (e.g., termination, correctness and completeness) that have to be proved for each solver, that is to say, the operational procedure specifically implemented for the solver has to be explained in a particular way and cannot be deduced from or generalised to another solvers. Moreover, it is not always easy to clarify which are the main properties of specific aspects of the solver (e.g., key aspects of the heuristics used) that are responsible for specific properties of the solver.

These drawbacks disappear when a generic approach is considered. For this reason, in the recent years, there are a number of proposals for general principles under which the main properties of the constraint solving algorithms can be explained and the operational behaviour of the different instances of the schema can be predicted. Moreover, with a generic solver a problem may be formulated in terms of constraints over several domains. This means that the final code will have a closer resemblance to the original formulation of the problem.

From this discussion it can be deduced that what is needed is a system with a constraint solver that can perform constraint solving independently of the nature and cardinality of the computation domain (as illustrated in Figure 1.2). Of course this system has to provide a unified framework that includes the usual CLP domains such as the integer, Boolean, sets and real domains.

A wider glass box schema. In current systems, many of the constraints have the control fixed by the system. These *black box* constraints provide very efficient tools for common constraint applications. However there are two practical problems with this approach.

- The constraint solving process is usually "hard-wired" in a low-level language so that it is difficult (sometimes, impossible) to build new solvers over new domains.

- A black box solver lacks adaptability when used for non-standard problems since the problem has to be coded with the built-in constraints provided by the system, that is to say, the degree of freedom the user has for coding a problem is bounded by the built-in constructs of the system.

To overcome this lack of flexibility, some constraint systems now provide *glass box* constraints. These allow new constraints to be defined by the user, although, in most cases, these are still restricted to particular predefined domains such as the integers. The advantage of a glass box system is that the user can define the constraints needed

Figure 1.2: A generic solver approach

for coding the best formulation of a problem. Furthermore, a glass box system usually allows the user to specify the constraint propagation schema itself often resulting in a more efficient program.

However, in the current glass box approaches the solvers are often restricted to just the built-in domains, usually the integers. This restricts the flexibility of this approach since as already discussed, in practice, problems are heterogeneous and often have a natural formulation which uses domains other than the built-in domains. Thus, current constraint systems lack expressiveness and the final code often bears no similarity to the initial problem statement.

Our solution to this problem consists in opening up the typical *glass box* approach to constraints on any computation domain. With this wider glass box approach, the user can define new domains, new constraints on these domains and, for each constraint, the

intended propagation behaviour of the domain values. We call this approach, the *two-dimensional glass box approach* since, as shown in Figure 1.3, it grows in two different directions (vertically, by defining planes which means the user defines new domains, and/or horizontally on any plane, by defining new constraints on each domain).

Figure 1.3: A two dimensional *glass box* approach: over constraints and over domains

To resolve all the problems outlined so far, and, hence combine the requirements for wider glass box systems and for generic solvers, we can see that what is needed is a two dimensional glass box generic CLP system.

A collaborative schema. As already commented, most of the existing CLP languages only provide support for solving constraints on specified domains, even if the system follows a glass box approach. This implies that the constraints, even if they are user defined, are usually restricted to just values in the given computation domain.

For example an arithmetic constraint such as $x + y = z$ must be defined so that all the elements x, y and z belong to the same domain (e.g., integer or real). However, as already remarked, many problems are most naturally expressed using heterogeneous constraints over more than one domain. Moreover, there exist constraints defined on multiple domains that require the collaboration of distinct domains by sending and receiving information to and from another different domain (e.g., $w = x > y$). This means that, even in the case that a CLP system provides a two dimensional glass box approach, again the formulation of real problems has to be artificially adapted to a single domain (i.e., one of the supported by the system).

A solution to this problem is in the concept of solver collaboration that involves both solver cooperation and solver combination. On one hand, solver cooperation allows constraints to be propagated from one computation domain to another thereby allowing information to flow between the different domains. On the other hand, solver combination enables either a mix of solvers (probably defined on different domains) or the generation of solvers defined on combined domains. In general, solver collaboration aims at overcoming two problems: a lack of declarativity of the solutions and a poor performance of the systems. However most of the existing work on this issue is dedicated to precise built-in domains and is neither flexible nor adaptable. A solution to this problem is in the concept of the glass box approach on the domains discussed above.

Again by linking all these requirements for CLP already discussed we conclude that what is needed is a generic system that allows the cooperation and combination of solvers defined on (possibly user defined) domains in which the constraint propagation schema can be specified at the user level.

An efficient schema. To our knowledge, most of the existing CLP systems do not satisfy all the requirements discussed so far. The main exception is the CHR language (Frühwirth, 1998) that allows users to define domains and constraints and interaction between the domains. Unfortunately, this solver is inefficient (Fernández and Hill, 2000a) and compares badly with most domain-dedicated constraint systems.

The main criticism of a generic approach is, although that efficiency is one of the main reasons for choosing CLP, the generic solvers rarely achieve the speed of the domain-specific solvers. However, a generic system has also important advantages. One is that any global improvement in the efficiency of the system can lead to a better performance in each of its possible instances. Moreover, from the theoretical point of view, a generic approach allows us to investigate general principles for constraint solving and therefore to devise general properties that are satisfied by all of its instances.

It is known that although most of the solvers are based on extensions of the consistency techniques in (Mackworth, 1977), the detailed reasoning differs for each domain. There are two basic forms of reasoning: domain reasoning and interval reasoning. Interval reasoning (which prunes just domain bounds) is more efficient than domain reasoning (which prunes the values in the domain). As a consequence, if we are concerned with efficiency, the interval reasoning appears to be the best basis for a generic

system.

By adding the interval reasoning to the previous set of requirements for our CLP system, we see that what is needed is a transparent, generic and collaborative interval-based CLP system. This thesis deals with the formalisation of a CLP system with these characteristics.

1.2 Chapter Summaries

This section presents the content of each of the remaining chapters.

Chapter 2: State of the Art of CLP

Previous work that is relevant to the present research is reviewed. After an introduction to CP and its historical development, it describes the CLP schema, with a brief overview of its theoretical framework and its applications. This is followed by a review of the most important glass box approaches used in CLP and other key instances of the CLP schema. The reader should realise that this chapter does not discuss all the publications related to the work of this thesis but only gives a global overview of the CLP foundations. We have preferred to discuss in each chapter the related work specific to that chapter. This helps keep each part of the document relatively independent.

Chapter 3: The Comparative Framework

This contains a detailed comparison for the Boolean and finite domains of eight popular but very different constraint systems under both the *black box* and *glass box* approaches. The comparison focuses on the efficiency and on specific aspects of the expressivity (concerning reified constraints and meta-constraints). This work has helped us choose the most appropriate constraint solving schema on which we could build our generic solver.

Chapter 4: A Constraint Propagation Schema

The theoretical foundations of our generic solver are defined. The chapter introduces an interval-lattice based structure over which constraints will be generically propagated and establishes the theoretical foundations for formalising the constraint propagation. An operational semantics for constraint propagation is provided and a number of interesting properties of the schematic procedure are studied. The chapter also shows how new domains can be constructed from the scratch and how existing domains can be combined.

This chapter constitutes the core of the thesis and establishes the basis for solver collaboration.

Chapter 5: A Solver Collaboration Schema

This chapter investigates how the framework for constraint propagation defined in Chapter 4 can provide a reduced one-directional form of solver cooperation. This is demonstrated by means of several non-standard examples. The theoretical framework is then extended with the concept of high level constraints that allow a two-way channel for the exchange of information between the domains and, hence, allowing complete collaboration between the solvers.

Chapter 6: A Constraint Branching Schema

This chapter completes our framework for constraint propagation for constraint solving by defining a schematic generic procedure for the branching of constraints. Important properties of the branching procedure including completeness, correctness and termination are established.

Chapter 7: The Practical Framework

To demonstrate the feasibility of our generic framework, a new logical language based on the theoretical work described in the previous chapters has been developed. In this chapter, the language is described and an overview of the main characteristics of a prototype implementation is given. A number of examples showing the potentialities of the system are provided.

Chapter 8: Conclusions

This chapter concludes the thesis by summarising the main results and discussing the major contributions. It also outlines some directions for further work.

Chapter 2

Basics of Constraint Logic Programming

Knowledge is of two kinds:
we know a subject ourselves, or we know
where we can find information upon it.

Boswell Life, vol.2, page: 407 (1775)
Samuel Johnson, 1709-84

2.1 Introduction

Constraint Programming (CP) (Marriot and Stuckey, 1998; Smith, 1995) has emerged as one of the most exciting paradigms of programming in recent decades. This chapter describes the state of the art of CLP ranging from its historical origin to its theoretical foundations. Since our objective is a generic glass box framework for solving constraints, the emphasis of this chapter is to provide a description of the most important instances of the CLP schema (that are also instances of our framework) as well as a presentation of the most important glass box approaches used in CLP. We include an introduction of the main foundations of the CLP schema.

2.1.1 Chapter Structure

The structure of the chapter is as follows: Section 2.2 gives a brief overview about what CP means and offers a quick guide to the different methods for solving constraint satisfaction problems. In Section 2.3, CLP is introduced by detailing its historical origins, the reasons for its appearance and the basic schema over which is founded. Section 2.3.6 provides a (traditional) discussion about the adequacy of using a black box or a glass box approach for solving constraints in the CLP schema. Sections 2.4

and 2.5 present a (non-exhaustive) summary of different languages that use different viewpoints to solve constraints under both the glass and black box approaches. Section 2.6 describes some important instances of the CLP schema. In Section 2.7, we introduce specialised constraints. These are built-in constraints that have been designed with the aim of providing either improved expressiveness or improved performance. Section 2.8 describes briefly further CLP languages that have contributed in some way to the development of CLP. Section 2.9 shows how the success of CLP has motivated the integration of constraints into other paradigms so as to better serve more other areas of application and briefly presents the foundations of the integration of constraints in these paradigms. The chapter ends by enumerating the contributions of the chapter.

2.2 Constraint Programming

2.2.1 Motivation

Constraint Programming has raised in recent years an increasing interest in the research community and has become one of the most interesting programming language paradigm ever invented. There are several reasons for this interest in CP. First, the strong theoretical foundations (Tsang, 1993) makes CP a sound programming paradigm. Secondly, CP is a heterogeneous field of research ranging from theoretical topics in mathematical logic to practical applications in industry. As a consequence, CP is attracting also widespread commercial interest since it is suitable for modelling a wide variety of optimisations problems, particularly, problems involving heterogeneous constraints and combinatorial search. Thirdly, CP is based on constraints which are basically relations between objects (e.g., variables constrained in a problem). It is the role of the programmer to define the constraints and the relations between the objects and entities managed in a program. However, with the usual imperative languages, not only does the programmer have to specify the relations or constraints between the objects (probably declared by the user) but also the means for computing over these relations.

EXAMPLE 2.1 *For instance, consider the relation*

$$x = y + z \tag{2.1}$$

In a traditional language a programmer cannot use this relation directly and it has to be coded depending on the known values for the involved variables x, y and z. For instance, in an imperative setting, if y and x are known then this relation is usually coded as follows

$$z \leftarrow x - y$$

Therefore, the programmer requires an additional effort since all the possible assignments derived from the relation (2.1) have to be maintained explicitly.

2.2.2 Constraint Satisfaction Problem (CSP)

CSP Modelling

A constraint is a relation maintained between the entities (e.g., objects or variables) of a problem. Constraints are used to model the behaviour of systems in the real world by capturing an idealised view of the interaction between the variables involved. The way in which this interaction is defined depends on the ability and skill of the programmer.

A CSP is a set of constraints involving a number of variables restricted to have values in a set of domains. In the CSP modelling, the CP paradigm has some analogies with respect to the traditional operational research (OR) approach. The step of modelling a problem usually consists of the following steps (Carro et al., 2000):

1. analysing the problem to be solved to understand which are its parts, that is to say, identifying the variables constrained;
2. determining the relation between the parts -the variables- by determining the conditions that hold among them and the domains involved in these relations, that is to say, determining the values that the constrained variables can take;
3. stating these relations as constraints in form of equations, disequalities, symbolic constraints or high level constraints (i.e. those involving primitive relations or constraints).

CSP Solving

Solving a CSP means the finding of a possible assignment (of values in the computation domains) for the constrained variables that satisfies all the constraints. Several cases arise:

- The CSP has one solution.
- The CSP has multiple solutions.

  EXAMPLE 2.2 *Consider a CSP with only one constraint $x < y$ where x and y are initially constrained to have values in the discrete domain $[0, 2]$. This CSP has three solutions:*

  $$x = 0,\ y = 1;$$
  $$x = 0,\ y = 2;$$
  $$x = 1,\ y = 2.$$

- The CSP has no solution. This can be due to to the impossibility to satisfy all the constraints at the same time (i.e., an *over-constrained problem*).

Solving practical problems usually involves three stages (Williams, 1993a; Williams, 1993b; Barth and Bockmayr, 1996):

1. Building a model of the problem.
2. Solving the model.
3. Understanding and analysing the solution.

The two first stages are treated throughout this document whereas the third one concerns more the area of program analysis.

CSP solving can be done by using different techniques ranging from traditional techniques to modern ones. For example, some approaches to solve a problem are in the area of OR, genetic algorithms, artificial intelligence (AI) techniques, rule-based computations, conventional programs and constraint-based approaches. Usually, CSP solving is understood as the task of searching for a single solution to the problem, although sometimes it is required to find the set of all solutions. Also, in certain cases, because of the cost of finding all solutions, the aim is just to find the best solution or an approximate solution within fixed resource bounds (e.g., in a reasonable time). Such kinds of CSPs are called *partial CSPs*. An example of a partial CSP is a *constraint optimisation problem* that assigns a cost to each solution and tries to find an optimal solution within a given time frame (Freuder and Wallace, 1992).

There exist different methods to solve a CSP. In general the methods to generate a solution for a CSP fall into four categories[1]:

1. The first category includes the variants of the backtracking search (in its various more or less intelligent versions) where one variable at a time, in a certain order, gets instantiated over the domain and more or less intelligent backtracking occurs when the current variable cannot be instantiated in any consistent way. This method is mostly used to solve search problems like the CSP. In this category we find the procedures *generate-and-test* and *standard backtracking*.

   Generate-and-test consists of generating each possible combination of the variables and then check if this combination satisfies all the constraints. This method only tests the constraint when an assignment to all the variables is done. Since the number of combinations is equal to the size of the Cartesian product of all the variables domains, this method is very inefficient as the size of the domains increases.

   Standard backtracking improves relatively the generate-and-test methods and consists of combining depth first search with chronological backtracking. The order in which variables and values are chosen is fixed and the variables are instantiated sequentially. If a partial instantiation violates any of the constraints, the latest instantiation is reconsidered. Of course this backtracking is able to eliminate a subspace from the Cartesian product of all variable domains. *Intelligent backtracking* improves standard backtracking by avoiding repeated fail due to the same reason[2]. It consists of backtracking directly to the last assignment for the variable that caused the fail.

[1] We have reformulated the three category set cited in (Ruttkay, 1998) to a four category set.
[2] This problem is usually called *thrashing*.

In general, these algorithms search systematically through the possible assignments of values to variables and, despite the fact that there is a guarantee to find a solution (if it exists) for the CSP or to prove its insolubility, they take a long time to do so.

Also another drawback of the search methods is that they works very badly when the given problem is highly redundant. However, in recent years new backtracking algorithms to solve this problem has been proposed. An exhaustive route by these algorithms is not an aim of this document and the interested reader is referred to (Kondrak and Van Beek, 1997).

2. The basic idea of the methods included in a second category is to remove redundancy before the search process. In general the process consists of removing, from the domains associated to the constrained variables, inconsistent values (that are those that can never be part of any solution) since an inconsistent value always leads to violate a constraint. This process reduces significantly the search and thus the amount of backtracking. These methods for solving CSPs are usually called *filtering algorithms*, *arc consistency algorithms* or also *propagation algorithms*. In general, the results are propagated through the whole constraint set and the process is repeated until a stable set is obtained.

   When all inconsistent values have been removed from the domains associated to a constraint c, then we say that this constraint c is *arc consistent*. An algorithm that deletes all inconsistent values for any domain of each constrained variable in the CSP, until all constraints are arc consistent, is usually called a *full arc consistent algorithm*. Unfortunately these algorithms are very costly. Therefore, there are other kinds of algorithms, called *partial arc consistent algorithms* that only delete some inconsistent values from the domains of the variables. Often, a well designed partial arc consistent algorithm deletes most inconsistent values at less cost than many full arc consistent algorithms.

   The main drawback of constraint propagation algorithms is that they are, often, incomplete in the sense that they are not adequate for solving a CSP, that is to say, they do not eliminate all the non-solution elements from the domains and as consequence, it is necessary to employ some additional strategy or another kind of algorithm to solve it. One complementary method to the propagation algorithms is that so called *constraint branching* that divides the variable domains and then continues with the propagation on each branch independently (see Section 2.2.3).

   In general CSPs may be very difficult to solve since they are NP-problems. For this reason, often working on small subsets of the variables (and related constraints) and never globally in the whole problem, leads to a significant change in the problem so that it is then, usually, much easier to solve. The algorithms following this approach are called *local consistency techniques* or *relaxation algorithms* (Montanari and Rossi, 1991; Mackworth, 1977). Often these algorithms provide an approximation to the solution, but in certain class of problems, they can also return the real solution. In this kind of problems the use of search it is

not necessary.

3. In a third category we include the *forward checking* and the *look ahead* algorithms that are methods embedding a consistency algorithm inside a backtracking algorithm.

   Forward checking prevents future inconsistencies by removing the values of the future variables (i.e., those variables not yet instantiated) which conflict with the value assigned to the current variable, that is to say, it tries to assure that each future variable has at least one consistent value with the current assigned variable (in fact with the already assigned variables). Of course, if the domain of a future variable is empty then a dead end is found since it is obvious that the current partial solution is inconsistent and a backtrack to another value of the current variable or to a previous variable is done. Trivially this means reduction of the search space by executing an *a priori* pruning of the search tree before trying (by backtracking) all the values for the future variables. Therefore, forward checking allows branches of the search tree that will lead to a failure to be pruned earlier than with simple backtracking.

   (Full) *looking ahead* is an extension of forward checking by assuring that each future variable has at least one consistent value with the other future variables. In fact this is a form to assure the arc consistency of the remaining problem to be solved. A variant is *partial looking ahead* that is in-between the forward checking and the look ahead algorithms. Basically, it consists of assuring the arc consistency for the remaining problem assuming a fixed ordering of the variables.

   Observe that the standard backtracking algorithm could also be included in this category since it can be seen as a mixture of pure generate-and-test and a fraction of arc consistency (i.e., the algorithm checks the validity of the constraints considering the partial instantiation of a number of variables).

4. The fourth category comprises the *structure driven algorithms* that exploit the structure of the constraint graph of the problem[3]. In this category we can find very different algorithms, even some that decompose the problem into subproblems which can be solved by methods from the previous two categories (Freuder and Hubbe, 1995). The decomposition of the original problem may be based on the checking of some features of the graph of the CSP which can be done by well known methods of graph theory (Sedgewick, 1984).

We are specially interested in the third category of algorithms to solve CSPs, in particular, in those algorithms combining constraint propagation and backtracking by means of branching techniques (that are described in the following).

[3] A CSP can be represented by different forms of graphs e.g., arcs may correspond to the constraints and nodes to the variables (Ruttkay, 1998).

2.2.3 Branching Techniques

As already mentioned, propagation algorithms are in general not enough to solve a CSP and, as consequence, it is very frequent that, when no more constraint propagation is possible, an additional strategy is employed to solve the CSP. A usual strategy is that called *branching* that is a (non-exclusive) technique of the CLP languages. Branching consists of dividing the domain(s) of some variable(s) when no more constraint propagation is possible and generating different computation branches in the search tree to further continue with the propagation on each of the branches independently. In general it is a controlled way of making choice to activate a new search in order to search for a solution. Obviously, this technique is implicitly employed in the backtracking search algorithms. Traditionally branching has been used in association with solvers over finite and discrete domains and, in this case, it is also called *enumeration* or *labeling*.

A branching strategy involves two steps of choice: the election of a variable from which the search is reactivated and the election of the value (or set of values) from which the independent branches of the search tree are explored. Efficiency is affected by the correct election of these processes. The first process is usually called *variable ordering* and the second one *value ordering*. In the variable ordering step a variable is decided to branch to next at a given node of the search tree. From the selection of a variable, new successor nodes corresponding to the possible values of the variable are generated. The value ordering step decides then which is the next node over which propagation begins again.

It has been shown that certain choices reduce the search space more drastically than others since the order in which variables and values are chosen has dramatic effects in the size of the search space and, as consequence, it has a great influence on the efficiency (specially if we are looking for the first solution; it has little effect when the search is for all solutions). Different choices change the shape of the search tree and, therefore, the order in which solutions are found. As consequence, the efficiency of a program depends to a large degree on the heuristic used for branching. Thus, the user is advised to consider good strategies that lead to find a (first) solution in a reasonable time. For these reasons, there has been an intense study about heuristics to judge how critical the variables and values to be chosen are. Most of these heuristics are based on the cardinality of the current domains for the variable and the already satisfied and remaining constraints.

EXAMPLE 2.3 *The* first fail *labeling uses a principle (Haralick and Elliot, 1980) which says that* to succeed, try first where you are the most likely to fail. *This principle recommends the choice of the* most constrained *variable which (for the finite domain) means choosing a variable with the smallest domain. This guarantees that inconsistent values will appear early in the search space and thus will affect other values of other variables faster than choosing another variable.*

A further refinement commonly used in first fail *labeling consists of choosing, in the case when there is more than one such recommended variable, the one that appears*

in the greatest number of constraints. This is called the most constrained heuristic *and guarantees that the value chosen will affect the maximum number of values of the constrained variables related directly with the chosen variable.*

In contrast to first fail*,* naive labeling *is a very simple labeling which chooses the left-most of a list of variables and then selects the smallest value in its domain.*

It is important to comment that the best ordering heuristic usually differs between the CSPs to be solved, and the same strategy can be more or less efficient dependent of the nature of the problem.

2.2.4 Some Review References

There are a number of reports and papers that introduces the main concepts of CP. For instance, (Smith, 1995) gives a basic account of CSPs and some of the algorithms used to solve them, including techniques commonly used in CP tools. Also (Barták, 1999) provides a survey of constraint programming technology and its application starting from the historical context, going on to show the interdisciplinary nature of CP and describing diverse aspects ranging from constraint satisfaction techniques to industrial applications. To find out more about CSPs and the methods to solve them, the reader is referred to (Kumar, 1992) and (Nadel, 1989) that present very complete surveys. Also in (Badía, 2000) we have developed a theoretical and practical comparison of search and arc consistency algorithms to solve CSPs.

2.3 Constraint Logic Programming

Initially, CLP (Benhamou and Colmerauer, 1993) was born as a mixture of two declarative paradigms: CP and LP (Lloyd, 1987). CLP adds to the LP declarativity the efficiency of constraint solving in such a way that CLP programs are both expressive and, in some cases, more efficient than other (even non-logical) programs.

2.3.1 A Brief History of LP Systems

The origin of the first LP language is not clear and seems to be shrouded in mystery. Back in the sixties, Alan Robinson established the foundations of the field of automated deduction by introducing the resolution principle, the notion of unification and a unification algorithm (Robinson, 1965; Robinson, 1971). Using the resolution method one can prove theorems of first-order logic. It seems that the first LP language, called Prolog, was invented by Alain Colmerauer and Phillipe Roussel at the University of Aix-Marseille in 1971 (Kowalski, 1988; Cohen, 1988) and was born of a project aimed not at producing a programming language but at processing natural languages; in this case, French (Colmerauer and Roussel, 1993). The project gave rise to a preliminary version of Prolog at the end of 1971. Also, the same year, (Kowalski and Kuehner, 1971) described a very interesting method of resolution, the *linear resolution*, that formed the basis of what we now call *SLD-resolution*. This form of resolution is more

restricted than that proposed by Robinson in the sense that only clauses with a limited syntax are allowed. Based on this issue, the preliminary version of Prolog used linear resolution with unification only between the head of the clauses. Later on, this method was proved to be complete when only Horn clauses were used (Kowalski, 1974). Prolog was first implemented in 1972 using ALGOL-W (Wirth, 1966). Since then, several characteristics were added to the system and the original version of Prolog was also implemented in other languages such as Fortran. These earlier implementations of Prolog ran slowly but they established the viability of Prolog.

In 1983, David H.D. Warren designed an abstract machine for the execution of Prolog consisting of a memory architecture and an instruction set (Warren, 1983). This meant a major breakthrough for the implementation of Prolog. Since then, this set of instruction has been accepted as the standard basis for the implementation of Prolog. This was the beginning of the future, modern and more efficient implementations of Prolog.

Despite the fact that it was designed originally for natural-language processing, it became one of the most widely used languages for artificial intelligence.

2.3.2 The CLP Schema

The CLP schema was introduced by Jaffar and Lassez in 1987 (Jaffar and Lassez, 1987). This schema was a formal framework, based on constraints, for the basic operational, logical and algebraic semantics of an extended class of logic programs. Although the CLP programs depend on the domain of application, the CLP schema was devised to create languages sharing the same evaluation mechanism.

The CLP schema comprises a family of languages. Given a particular domain and a constraint solver, the schema defines a language for writing programs and a mechanism for evaluating goals and programs written in this language. The constraint domain details the primitive constraints in the language, while the solver is used in the evaluation of goals.

As already claimed, CLP combines the declarativity of LP with the efficiency of CP. In the following we present an global overview of the CLP schema. The operational semantics and the declarative semantics of LP are simple and it is expected that the reader is familiar with them.

The basic idea in CLP is to replace the classical LP unification by constraint solving on a given computation domain. This idea gave rise to the CLP($\mathcal{X}$) schema that was described in (Jaffar and Lassez, 1987). In this schema $\mathcal{X}$ is the computation domain over which constraints are solved. Different instances of $\mathcal{X}$ (e.g., with reals, integers, sets, Booleans, etc.) generate different instances of the CLP($\mathcal{X}$) schema.

In each of these instances the underlying logic programming language is extended with a set of operations and structures to be used over the computation domain that can be directly manipulated by the user. That is to say, the user can make use of operations on the values of the computation domain (e.g., sums on the integers and intersection on sets) and relations between them (e.g., equational expressions on reals). These relations are called *constraints* and usually the CLP system groups them in a

structure called a *constraint store*.

The CLP($\mathcal{X}$) schema is also characterised by the introduction, in the logic language, of a mechanism for the resolution of the constraints. This mechanism, called the *constraint solver*, constitutes a decision procedure able to test whether a constraint or a set of constraints is satisfiable. The constraint solver replaces standard LP unification by an algorithm that allows to decide the satisfiability of constraints in the system where the satisfiability check means to decide the consistency of the constraints. If the constraint solver can decide the satisfiability of all the constraints then we say that it is *complete*. However, most of the existing solvers are not complete and they encapsulate some kind of constraint propagation algorithm (see Section 2.2.2). Observe that the faster the underlying algorithm built-in inside the solver, the better the performance of the constraint solver is. The efficiency of the constraint solver algorithm is clearly conditioned by the nature of the computation domain.

One of the main features of the constraint solver of a CLP system is the *incrementality of constraints*. This property consists of avoiding computing the satisfiability of constraints whenever a new constraint is added. When executing CLP programs, usually new constraints are added during the execution, for instance via the recursion. After each addition of a constraint, the solver is invoked to check consistency. Incremental solving means that the solver checks the consistency of the added constraints with respect to the existing constraints in the main store without checking it again for the constraints in the store. The incrementality must cooperate with backtracking. If a solver detects inconsistency after the addition of an incremental constraint, all constraints added since the last choice point have to be removed. Then, computation is restarted and new constraints (and variables) can be added to the store. From the incrementality property, another interesting feature of CLP is deduced: during execution new variables and constraints can be created dynamically what means that neither the number of variables nor the number of constraints is known before and during computation.

In a general context we can say that CLP can be viewed as the incorporation of constraints and constraint solving methods in a logic-based language. The main difference between LP and CLP relies then in the operational interpretation of the constraints rather than in their declarative interpretation.

The critical idea of the CLP schema is that a logic-based programming language, its operational semantics, its declarative semantics and the relationship between these two semantics can be parameterised by a computation domain $\mathcal{X}$ and its constraints. The interested reader is referred to (Jaffar and Maher, 1994) for a more detailed description of the semantics of CLP.

2.3.3 Motivations for CLP

The success of CLP programs is not only result of the improved performances with respect to LP programs but it relies on the functionality of them. CLP is adequate to solve problems that LP cannot solve.

EXAMPLE 2.4 *Consider the simple logic program*

```
less_than_three(X):-X < 3.
```

A query such as

```
:-less_than_three(Y).
```

returns an instantiation error in standard LP since Y is not ground before the call. However, when this goal is told in CLP then the answer $Y < 3$ is returned.

2.3.4 C(L)P Applications

A large variety of real applications involve relations and constraints among the entities of a problem. For this reason, CP has shown a commercial interest since is oriented mainly to the specification of constraints and relations among the objects and entities involved in a problem (or program). The first CP languages were not totally successful since techniques to solve the constraints were integrated in the traditional languages in a *ad hoc* way. In definitive, CP languages did not show sufficient expressiveness and were not powerful enough to solve constraints. However, in the last decades new CP languages have emerged with new characteristics. Mainly, constraints are integrated in the host language and the user is allowed to use constraints on a high level whereas constraints are solved by means of powerful incremental constraint solvers. The whole system, that is, the host language and the integrated solver, is a true programming language that is adequate to solve a wide ranges of applications.

Currently CP is a powerful tool for modelling many real-worlds problems since, because of its inter-disciplinary nature, it combines and exploits ideas from a number of fields such as AI, discrete mathematics, OR, symbolic computations, programming languages, robotics and genetics among others. The heterogeneous nature of CP is adequate for a wide ranges of applications. Proof of it is the number of interesting and diverse applications described in (PACT'96, 1996; PACT'97, 1997; PAPPACT'98, 1998; PACLP'99, 1999; PACLP'2000, 2000).

Specifically regarded to CLP, CLP offers facilities for problem modelling, constraint propagation and search and has been proved to be useful in the solving of a wide range of applications. For instance, CLP has been applied to applications traditionally solved by OR techniques like combinatorial search (Dincbas et al., 1990), cutting stock (Dincbas et al., 1988a) and scheduling problems (Curtis et al., 2000). Other examples of areas where CLP has been proved to be useful are in fault diagnosis in circuits (Azevedo and Barahona, 2000; Simonis and Dincbas, 1987), neural networks (Kok et al., 1996) and other industrial applications (Carlsson and Brindal, 1993). Moreover, the propagation in CLP is exploited in applications involving user feedback and graphical interfaces such as in visualising relationships in biological data (Gilbert et al., 2000). More exotic applications include control of handling robots (Sato and Aiba, 1993a) and reconstruction of the original DNA sequence from the fragments of an enzyme partition (Yap, 1993). For more information, (Wallace, 1996) and (Simonis, 1995) give a number of examples of industrial applications of CLP.

2.3.5 Some Review References

We have introduced CLP in a global way. The interested reader in more specific CLP issues is encouraged to read the texts cited below.

From an introductory point of view, (Lassez, 1987) and (Frühwirth et al., 1993) provide an informal introduction to CLP.

From an historical view, (Cohen, 1990) gives a short introduction to CLP languages.

From the overall view, (Csontó and Paralič, 1997) and (Jaffar and Maher, 1994) are a global survey about CLP by giving systematic descriptions of the major trends in terms of common fundamental concepts (covering parts such as theory, implementations and applications among others).

From the teaching view, (Carro et al., 2000) serves as an introductory course to CLP mainly directed to industrial programmers with little previous experience with CP.

From the semantics view, (Jaffar et al., 1998) presents the semantics foundations of CLP in a self contained paper.

2.3.6 Black box vs. Glass box

Since its appearance, CLP has raised a discussion about if the constraints in a CLP system should be totally transparent (i.e., *the glass box approach*) or hidden (i.e., *the black box approach*) to the user. This discussion gives rise to controversy since both approaches have advantages and disadvantages. In this section we try to clarify this point.

The Black Box Approach. In the beginning, CLP provided specific built-in constraints to solve specific applications. These constraints are black boxes from the user point of view in the sense that the user can make use of them but does not need to understand in full detail their execution behaviour.

There have been two main reasons for the provision of black box constraints. Firstly, they are coded for specific applications so that they often allow a succinct way to model a wide set of problems. The second reason is the efficiency that these constraints provide to the standard solvers. The control of the black box constraints if fixed by the system that enables very efficient implementations of them. The internal constraint solver of the system uses specialised propagation mechanisms for these constraints leading thus to a major efficiency in the solving of well known and complex problems. Some examples of these specific-application and complex constraints are the *cumulative* constraint (Aggoun and Beldiceanu, 1992), the *all different* constraint (Régin, 1994) and the *element* constraint (Dincbas et al., 1988a) (see Section 2.7).

Black box constraints have also evident disadvantages. First, they are built-in in the system and coded internally in very complex manners since they are based on specialised built-in propagation rules. Thus, it is very difficult to understand their operational behaviour. Even for larger applications, it can be impossible to understand it. Despite this is part of the declarative nature of LP (i.e., the users know what to

do but not how to do it), when speaking about CLP, there are many people in the constraint community believing that the user does not need to know how a program is executed, but at least should be able to understand the exact behaviour of the execution, for instance to debug and/or correct their programs. Another drawback is that these complex constraints lack adaptability for being use in non-standard problems. As told above, they are provided for specific modelling of problems so that they are not useful for applications that do not follow an standard schema. Thus these constraints do not provide flexibility to be used out from their area of application.

The Glass Box Approach. To overcome the lack of flexibility and applicability of black box constraints, it was demanded that "constraint solvers must be completely changeable by users" (p. 276 in (Aiba et al., 1988)). Since then, a new proposal was made to allow for more flexibility and customisation of constraint systems. This proposal is called *glass-box* or even *no-box* approaches. This approach allows the user to define new constraints in terms of the primitive constraints provided by the system. The advantages are clear. In practice, there are many constraints that are not specific to standardised applications and the users can define their own *glass box* constraints specialised for particular applications. Also, *glass box* constraints are useful in the re-utilisation of code since they are easily adapted to solve similar problems to those problems for which they were designed. Moreover, the user understands totally the operational behaviour of their constraints so that is very easy to modify them in order to find for the most adequate constraint solving for a problem or modify them to find a correct behaviour of them. This global understanding of the process allows for general optimisations, as opposed to the many local and particular optimisations hidden inside the black box constraints.

Two main criticisms to the glass box approach can be done. First, as glass box constraints are not specific to particular applications, their efficiency depends directly on how the user defines them. For this reason, the user should use black box constraints whenever possible and when the application allows them. Also, even in the cases that a (careful) use of user-defined constraints can always be recommended, in order to enforce more homogeneous grains of information and favour the main streams of propagation, one has to be cautious about correctness and completeness of the definition of the glass box constraints.

A fuzzy frontier. The distinction between a glass or a black box system for CLP is not always clear. For instance, a black box system can provide primitive constraints to allow the user to define new constraints. However, this does not mean that this system is a glass box system. The gap between a glass or a black box system is in the level on which a user is allowed to define constraints. *Glass box* languages (Van Hentenryck et al., 1994) are considered those languages which provide very simple and primitive constraints in which the propagation schema can be formally specified. These constraints can then be used to construct specialised high level constraints suitable for the application. Alternatively, *black box* languages are those which provide a wide range of

high level constraints whose implementation is hidden from the user. These constraints perform specific tasks very efficiently. In these languages, it is hard for a user to add new constraints since such constraints have to be defined at a low level requiring a detailed knowledge of the implementation.

Conclusion. It follows from this discussion that what is needed is more efficient glass box systems. Of course, these systems can also provide specific-application constraints allowing their use on standard problems. However, the main power of these systems consists of the provision of efficient primitive constraints so that the user can combine them and reuse them to solve non-standard CSPs. This combination of efficient constraints should lead to efficient high level and complex constraints.

2.4 Main Glass Box Approaches

As already declared in Chapter 1, our interest is in the glass box systems. The primitive constraints provided by such systems are seen as propagation rules, that is, rules for describing arc consistency propagation. From this perspective, there have been two main separate developments for the provision of glass box constraints. the first was developed in the constraint system clp(FD) (Codognet and Diaz, 1996a) which is designed for the finite domain of integers and based on a single constraint often referred to as an *indexical*. This approach, which allows the user to define and control the constraint propagation within the finite domain of integers via *indexicals*, is very efficient since the implementation uses a simple interval narrowing technique which can be smoothly integrated into the WAM (Aït-kaci, 1999; Diaz and Codognet, 1993). As a result, clp(FD) is now integrated into mainstream CLP systems such as SICStus Prolog (Carlsson et al., 1997) or IF/Prolog (If/Prolog, 1994). The other development is the CHR language (Frühwirth, 1998) which allows the user to define the constraint propagation via *constraint handling rules*. These rules enable the user to create new domains and define their solvers and any interaction between them.

In the following we describe these two glass box systems.

2.4.1 The Indexical Approach

The idea of *indexical* was originally designed for the framework of concurrent logic languages and taken from (Van Hentenryck et al., 1991). This paper broke the monopoly of the black box approach originally presented in all the CLP systems. The idea consisted of reducing a linear expression to a set of lower level constraints called indexicals. Indexicals are primitive constraints, provided in the CLP model for FD, of the form X *in* r where X (i.e., the *constrained variable*) is a FD variable and r is an expression to be evaluated in the domain of the sets of integers.

Indexicals allow to specify the intended propagation mechanism and from them it is possible to define new high level constraints. As the operational behaviour of indexicals is simple, an indexical-based CLP system follows a glass box approach.

Although initially thought for the concurrent setting, the constraint X *in* r supposed a good basis for an abstract machine for CLP on FD. A further step was to integrate into the WAM architecture the constraint X *in* r (Diaz and Codognet, 1993). This extension was specifically designed for the finite domain and showed how complex constraints could be compiled in efficient X *in* r expressions. The result was the system clp(FD) (Diaz, 1994; Diaz, 1995; Codognet and Diaz, 1996a) that allows to solve constraints defined on FD (i.e., the integer and Boolean domains).

The clp(FD) system. In this system a *range* is a subset of

$$\{-infinity, ..., -1, 0, 1, ..., infinity\}$$

where *infinity* and $-infinity$ are particular integers denoting the greatest value and the lowest value that a variable can take (i.e., the top and bottom elements). The notation a..b is used as a shorthand for the set

$$\{x \mid a \leq x \leq b\}$$

(that is, [a,b] in interval notation). In addition to some operations on ranges (e.g., union, intersection, etc.), other operations, called *pointwise operations* (i.e., +, -, *, /) are defined between a range r and an integer i and return the set obtained by applying the operator on each element of the range.

The clp(FD) system is based on domain constraints and indexicals. A *domain constraint* is an expression of the form X *in* r that forces X (the constrained variable) to have values in a range r. The constraint $X = n$ where n is a natural number is treated as a shorthand for the constraint X *in* $n..n$. In this case we say that n is the ground value for X.

An *indexical* is an expression of the form X *in* I where X is a FD variable and I is an expression, called *indexical expression*, to be evaluated in the domain of the integer sets. An indexical expression usually contains one (or more) of the following terms (called *indexical terms*): $min(Y)$, $max(Y)$, $dom(Y)$ or $val(Y)$ where Y (i.e., the *indexed variable*) is any FD variable distinct from X.

The clp(FD) system has one unique primitive constraint of the form

$$X \ in \ R$$

where R is a range or an indexical expression. To distinguish R from a range r, we will say that R is an *evaluated range*. The (simplified) syntax of the primitive constraint is shown in Table 2.1.

Constraint solving is based on the concept of constraint store and on the evaluation of indexicals.

A *constraint store* is a set S of primitive constraints. For any FD variable X, we write D_X^S to express *the domain of* X *in* S that is defined as the intersection of all ranges r associated to X in S, that is to say, all ranges r such that a domain constraint of the form X *in* r exists in S. This intersection process is known as *constraint narrowing*.

Table 2.1: Basic syntax of the constraint X *in* R in clp(FD)

constraint ::=	X *in* eRange	(constraint)
eRange ::=	term..term	(evaluated range)
	{t}	(singleton)
	eRange $\vee$ eRange	(union)
	eRange $\wedge$ eRange	(intersection
	-eRange	(complement)
	eRange $\bullet$ ct	(pointwise operation: $\bullet \in \{+, -, *, /\}$)
term::=	ct	($ct \in$ *Integer*)
	infinity	(top element)
	-infinity	(bottom element)
	$min(X)$	(minimum indexical)
	$max(X)$	(maximum indexical)
	$val(X)$	(ground indexical)
	$dom(X)$	(domain indexical)
	term $\bullet$ term	($\bullet \in \{+, -, *, /\}$)

The store is inconsistent if the domain of some variable is empty; otherwise, it is consistent.

If S is a constraint store and Y is a variable constrained in S by some domain constraint, then the values in S of $min(Y)$, $max(Y)$ and $dom(Y)$ are defined to return the lower bound of D_Y^S, the upper bound of D_Y^S and D_Y^S respectively. Indeed, if there exists a ground value n for Y in S then the value in S of the indexical term $val(Y)$ returns n; otherwise the evaluation of this function is delayed. If I is an indexical expression, evaluating I with respect to S means to return the expression I_S that results from replacing all occurrences of indexical terms in I by their value in S.

By simplifying the process, basically constraint solving in the clp(FD) system is executed incrementally as follows: there exists one main constraint store S that initially contains one domain constraint X *in* $-$ *infinity..infinity* for each constrained variable X in the problem. The constraints are then added to S incrementally. Each time that a new indexical X *in* I is added to S or constraint narrowing in S reduces D_Y^S where Y is any variable indexed by some indexical X *in* I belonging to S, the domain constraint X *in* I_S is added[4] to S where I_S is the evaluation of I in S. The process begins again by executing constraint narrowing. The process generating new constraints is called *constraint propagation*. The whole process terminates when S becomes inconsistent or when no more constraint propagation is possible because either constraint narrowing reduces no domain of the constrained variables or no indexical term is affected by

[4]Assuming that I_S is evaluated to a range.

constraint narrowing.

EXAMPLE 2.5 *Consider the following constraint store resulting from a previous constraint narrowing step*

$$S \equiv \{\ X\ in\ 4..10,\ Y\ in\ 3..20\ \},$$

and suppose that the indexical

$$Y\ in\ 0..max(X) - 1$$

is now added to S*. Then,* $0..max(X)-1$ *is evaluated to* $0..9$ *since the value of* $max(X)$ *in* S *is 10. As consequence, the indexical* $Y\ in\ 0..max(X) - 1$ *is propagated (i.e., evaluated) to the constraint* $Y\ in\ 0..9$ *that is added to* S*. As consequence of further constraint narrowing,* S *becomes*

$$S \equiv \{\ X\ in\ 4..10,\ Y\ in\ 3..9,\ Y\ in\ 0..max(X) - 1\ \}.$$

and the process terminates since $max(X)$ *is no more affected.*

The primitive constraint $X\ in\ R$ can be used to specify the declarative description of the problem.

EXAMPLE 2.6 *The program of Example 2.4 can be coded as*

```
less_than_three(X) :- X in -infinity..2.
```

The constraint $X\ in\ R$ embeds the propagation mechanism of the whole system. It is used not only to specify the declarative description of the problem to solve, but also to provide the propagation mechanism of the constraints i.e., the exact description of how constraints should be propagated and the way the domains are pruned. Constraint propagation is specified via the correct use of the indexical terms. Of course, by using the primitive constraint $X\ in\ R$ it is possible to build high level constraints on which the propagation mechanism is completely specified.

EXAMPLE 2.7 *Consider the constraints* $X = Y + C$ *and* $X \neq Y$ *where* X, Y *are FD variables and* C *is an integer value. These constraints are user-defined in terms of indexicals as*

$$\begin{aligned} X = Y + C \Leftrightarrow\ & X\ in\ min(Y) + C..max(Y) + C, \\ & Y\ in\ max(X) - C..max(X) - C. \end{aligned}$$

$$\begin{aligned} X \neq Y \Leftrightarrow\ & X\ in\ -val(Y), \\ & Y\ in\ -val(X). \end{aligned}$$

Different schemas of propagation are devised by using different indexicals terms. For instance, in the constraint $X \neq Y$, the indexical $X \ in \ -val(Y)$ is delayed until Y is bound. This means propagation is executed in a *forward checking* manner. On the other side, constraint $X = Y + C$ is propagated via a *partial look ahead* schema, since only changes in the minimum and maximum of X and Y are propagated. A *(full) look ahead* schema would propagate the whole domain of variables by using the indexical term *dom* as shown below. See (Van Hentenryck, 1989) for more information.

EXAMPLE 2.8

$$X = Y + C \Leftrightarrow X \ in \ dom(Y) + C,$$
$$Y \ in \ dom(X) - C.$$

2.4.2 Constraint Handling Rules

Constraint handling rules (CHRs) (Frühwirth, 1994; Frühwirth, 1998; Frühwirth, 1999) are a proposal to allow more flexibility and application-oriented customisation of constraint systems. CHRs are a declarative language extension especially designed for writing user-defined constraints and constitute essentially a committed-choice language consisting of multi-headed guarded rules that rewrite constraints into simpler ones until they are solved. A CHR language allows "multiple heads", i.e., conjunctions of constraints in the head of a rule, that are a feature that is essential in solving conjunctions of constraints. With single-headed CHRs alone, unsatisfiability of constraints could not always be detected (e.g., $X < Y, Y < X$) and global constraint satisfaction could not be achieved.

As a special purpose language, CHRs extend a host language with (more) constraint solving capabilities. Auxiliary computations in CHR programs are directly executed as host language statements. To keep this section essential and self-contained, we will not address host language issues here.

A constraint is considered to be a distinguished, special first-order predicate (atomic formula). We use two disjoint sorts of predicate symbols for two different classes of constraints: one sort for built-in (predefined) constraints and one sort for CHR (user-defined) constraints. Built-in constraints are those handled by a predefined constraint solver that already exists in the host language. CHR constraints are those defined by a CHR program where a CHR program is a finite set of CHRs. Since host language statements that appear in CHRs must be declarative, we can consider them as built-in constraints in this section (with a rather incomplete solver, the host language).

In the following we give an overview of CHR syntax. For more information about their semantics as well as soundness and completeness results see (Abdennadher et al., 1999; Abdennadher, 1997). There are three kinds of CHRs:

- the *simplification* CHR which has the form

$$label \ @ \ H_1, \ldots, H_i \Leftrightarrow G_1, \ldots, G_j \ \mid B_1, \ldots, B_k;$$

- the *propagation* CHR which has the form

$$label\ @\ H_1, \ldots, H_i \Rightarrow G_1, \ldots, G_j \ \mid B_1, \ldots, B_k;$$

- and the *simpagation* CHR which has the form

$$label\ @\ H_1, \ldots, H_l \backslash H_{l+1}, \ldots, H_i \Leftrightarrow G_1, \ldots, G_j \ \mid B_1, \ldots, B_k$$

where $i > 0, j \geq 0, k \geq 0, l > 0$. The multi-head $H_1, \ldots, H_i$ is a non-empty sequence of CHR constraints, the guard $G_1, \ldots, G_j$ is a sequence of built-in constraints and the body $B_1, \ldots, B_k$ is a sequence of built-in and CHR constraints. *label* is simply an identifier for the rule. Simplification replaces constraints $H_1, \ldots, H_i$ by simpler constraints $B_1, \ldots, B_k$, provided conjunction of guards $G_1, \ldots, G_j$ can be proved. Propagation adds to the store new constraints $B_1, \ldots, B_k$ which are logically redundant with respect to constraints $H_1, \ldots, H_i$ but may cause further simplification (again provided the conjunction of guards $G_1, \ldots, G_j$ can be proved). Simpagation is an abbreviation of the simplification rule

$$label\ @\ H_1, \ldots, H_l, H_{l+1}, \ldots, H_i \Leftrightarrow G_1, \ldots, G_j \ \mid B_1, \ldots, B_k, H_1, \ldots, H_l.$$

EXAMPLE 2.9 *Consider the constraint $X \geq Y$. Then, the following CHRs define a partial order constraint $\geq$.*

$$reflexivity\ @\ X \geq Y \Leftrightarrow X = Y \mid true.$$
$$antisymmetry\ @\ X \geq Y,\ Y \geq X \Leftrightarrow X = Y.$$
$$transitivity\ @\ X \geq Y, Y \geq Z \Rightarrow X \geq Z.$$

'true' represents empty sequences of CHRs. These rules defined the propagation of the constraint $\geq/2$. For instance, the reflexivity *rule declares that if the store implies $X = Y$ then a constraint such as $X \geq Y$ can be simplified to true (and thus be removed from the store).* Antisymmetry *rule declares that if two constraints $X \geq Y$,$Y \geq X$ are implied by the store, then they can be replaced by the simpler constraint $X = Y$. Last,* transitivity *rule adds the constraint $X \geq Z$ to the store whenever two constraints $X \geq Y$, $Y \geq Z$ belong to it.*

Repeatedly applying CHRs incrementally simplifies and, possibly, solves the user-defined constraints.

CHRs have been integrated in a number of CLP languages, providing them a glass box approach and have been used to encode a wide range of constraint solvers, from standard domains such as the finite and set domains to new domains such as feature trees and domains for the terminological and temporal reasoning (Frühwirth, 1998). Because of their flexibility, CHRs have also been used to model real applications (Frühwirth and Brisset, 1997; Frühwirth and Brisset, 1998; Frühwirth and Abdennadher, 2001).

2.5 Some *Black box* Languages.

In this section, we described briefly the constraint solving mechanism of a number of CLP *black box* systems. In fact, these are glass box systems from an strict definition since they allow the user to define new constraints. However, all of them require a (detailed) knowledge of their implementation in order to specify the propagation mechanism of the new constraints. For this reason, in the following, we classify these systems as black box systems.

Oz (Smolka, 1995) is a new language[5] combining functions with relations so that it has the potential for extra expressiveness in the constraint solver. It provides algorithms to decide the satisfiability and implications for basic constraints which take the form $x = n,\ x = y$ or $x :: D$ where x and y are variables, n is an non-negative integer and D is a finite domain[6]. The basic constraints reside in the constraint store. Non-basic constraints, such as $x + y = z$, are not contained in the store but are imposed by *propagators* (Müller and Würtz, 1996). An Oz *propagator* is a computational agent which is posted on the variables occurring in the corresponding domain. It reads the constraint store and tries to narrow the domains posted there by amplifying the store with basic constraints.

EXAMPLE 2.10 *Suppose there is a constraint store containing the domain variables* X, Y *with the domain* $\{1, \ldots, 10\}$. *The propagator for* $X + Y = 5$ *narrows the domain for both* X *and* Y *to* $\{1, \ldots, 4\}$. *The propagator* $X + Y = 5$ *is said to constrain the variables* X *and* Y. *Adding the constraint* $Y = 1$ *narrows the domain of* Y *to* $\{1\}$ *and the domain of* X *to* $\{4\}$.

Propagators are provided by both glass and black box languages. However, in glass box languages, the primitive constraints (e.g., indexicals and CHRs) are the basic components and the propagators are constructed from them. On the other hand, in black box languages, there is no means of specifying directly the constraint propagation so that propagators such as $+$ or $-$ are themselves primitives and are determined solely by their operational semantics.

ECL^{i}PS^e (Aggoun et al., 1995) includes the traditional finite domain constraints that are now being incorporated in many logic programming systems. It also supports writing further extensions such as new user-defined constraints or complete new constraint solvers such as CHR. These extensions are based on a mechanism of suspension and waking of goals provided by ECL^{i}PS^e. To make such an extension, the user needs a good knowledge of the underlying system and this is the reason why ECL^{i}PS^e is not catalogued as a *glass box* language.

[5] Currently, there exists a distributed version called *Mozart*.

[6] The constraint $x :: D$ restricts x to have values in the domain D.

Ilog SOLVER (Ilog, 1995) is a C++ library for constraint programming so that the underlying data and control structures must be defined in C++. Thus it is not strictly a CLP system. However, as it uses the CLP approach and is a popular commercial system for solving constraint satisfaction problems, we include it in this section.

In Ilog SOLVER, a constraint is either an object or a Boolean expression with values false (**IlcFalse**) or true (**IlcTrue**). The actual value depends on the satisfiability of the constraint: if the constraint cannot be violated, then the expression is bound to **IlcTrue** and if the expression cannot be satisfied then it is bound to **IlcFalse**. These expressions can themselves be constrained or combined with logical operators *or*, *and* and *not* to create more complex constraints.

When a constraint is posted (by means of the function **IlcPost**), the constraint is used immediately to reduce the domains of the constrained variables that it involves. Backtracking is provided by combining non-deterministic elements.

B-Prolog (Zhou, 1997) is based on a new abstract machine called ATOAM (Zhou et al., 1990; Zhou, 1994) (yet Another matching Tree Oriented Abstract Machine). Since most of the existing Prolog systems are constructed from the basis of the WAM, B-Prolog is a system that provides an alternative to the WAM approach. B-Prolog has been proved to be often significantly faster than most WAM-emulator-based Prolog systems (Zhou and Nagasawa, 1994). This system provides new facilities for constraint solving and presents some clear differences with respect to other clp(FD) systems.

As with ECL^{i}PS^e, this system provides a set of traditional FD predicates such as arithmetic or Boolean constraints and a set of primitives to process the domain variables. Note that this set of built-in constraint predicates is smaller than that provided by ECL^{i}PS^e.

It is also possible to define constraints by means of delay clauses (that are a special kind of construct provided by this system) although the user needs some knowledge about the underlying mechanism of the language (Section 7.6 describes in greater depth the mechanism of delay clauses in B-Prolog).

2.6 CLP Instances: Dependent-Domain Reasoning

As declared in Section 2.3.2, the CLP schema is parameterised with respect to the underlying computation domain over which constraints are solved. Constraint solvers are specific to the computation domain so that different domains highlight different reasoning mechanisms for constraint solving. In this section we described some of the most important instances of the CLP schema, although the list is not exhaustive of course.

2.6.1 The Finite Domain

Of the domains for CLP, the Finite Domain (FD) (Van Hentenryck, 1989) is one of the most and best studied since it is a suitable framework for solving discrete constraint

satisfaction problems. The importance of the CLP languages based on the finite domain is their impact in the industry since a lot of problems in the real life involves variables ranging on discrete domains. This means that CLP languages for FD are appropriate to solve many real-world industrial problems.

FD is particularly useful for modeling problems in areas like operation research, hardware design or artificial intelligence. Problems such as scheduling, planning, packing, timetabling can be modelled by FD variables and, as a consequence, most of the CLP systems provide substantial FD libraries.

The constraint propagation algorithms in CLP(FD) are usually called *consistency techniques* or *filtering algorithms* and they were born as an alternative to the inefficiency of the procedures *generate-and-test* and standard backtracking of LP (see Section 2.2.2). Consistency techniques are based on the idea of *a priori* pruning i.e., constraints are used to reduce the search space before a fail is found, and allows the reduction of the number of backtrackings as well as constraint checks. These techniques originated from Waltz's filtering algorithm (Waltz, 1972) and the algorithm of the problem solver REF-ARF (Fikes, 1968). Later on, these works were developed and extended in (Gaschnig, 1974; Mackworth, 1977; Freuder, 1978) among others. For an overview about the pioneering constraint satisfaction algorithms see (Nadel, 1989; Prosser, 1993).

Consistency techniques of CLP systems for finite domains have been described in an excellent way in (Van Hentenryck, 1989). Here it is proposed to embed the consistency techniques inside a LP language without loosing the natural formulation of logic programs. This embedding avoids to write logic programs using consistency techniques which would lead to less natural formulation. By embedding the techniques inside the system, the programmer does not need to care if the tree pruning is done *a priori* (i.e., by using the consistency techniques) or *a posteriori* (i.e., by applying the traditional backtracking).

The first CLP language for FD was the CHIP language (Dincbas et al., 1988b; Van Hentenryck, 1988). The propose of the CHIP language was to solve, efficiently and flexibility, a large class of combinatorial problems (Dincbas et al., 1990). In fact, CHIP was not only focused on the finite domain but involved three different domains of computations: the finite domain, Boolean terms and rational terms. Each of these domains has its own resolution mechanism coded internally in the CHIP system. For instance, in the rational domain, an Simplex-like algorithm was employed whereas Boolean unification was used in the Boolean domain. Proof of the success of CHIP is the fact that it was taken as the guide language to show the potentialities of the consistency techniques in (Van Hentenryck, 1989).

Since CHIP, constraint propagation in FD have been widely studied and, certainly, improved. The indexical approach (see Section 2.4.1) has been adopted by most of the CLP languages on FD. In particular, the major successful CLP system on FD is clp(FD) (also described in Section 2.4.1). Its particular glass box approach and the efficiency shown in the solving of real problems have popularised this system. Moreover, main existing CLP systems provide libraries for FD constraint solving based on the syntax of the clp(FD) system (shown in Table 2.1). Most of the main results of the clp(FD) system were published in (Diaz, 1995).

Recently, an alternative approach to the indexical model is proposed and integrated in the language B-Prolog (Zhou and Nagasawa, 1994; Zhou, 1996) that was briefly described in Section 2.5. This system provides specific constructs for constraint solving in FD and has shown to be acceptably efficient (Fernández and Hill, 2000a). However, currently it is not widely popularised yet. Section 7.6 describes in greater depth the special constructs for FD constraint solving in B-Prolog.

The interested reader is referred to (Henz and Müller, 2000) that gives an overview of constraint programming over FD for solving combinatorial problems.

2.6.2 The Continuous Domain

Many applications involve numerical computations in non-discrete domains i.e., in the real domain. The first CLP system that allows constraint solving on the real domain was the CLP($\Re$) system (Jaffar et al., 1992b). This system is an instance of the CLP schema and thus its operational semantics are very similar to Prolog. As usual in CLP, unification is replaced by constraint satisfaction and, in the particular case of CLP($\Re$), by constraint solving on the domain of uninterpreted functors over real arithmetic terms. CLP($\Re$) is a constraint logic programming language with real-arithmetic constraints. The implementation contains a built-in constraint solver which deals with linear arithmetic and contains a mechanism for delaying non-linear constraints until they become linear (Jaffar and Michaylov, 1987). From the implementation point of view, this delay mechanism is described in (Jaffar et al., 1991) whereas (Jaffar et al., 1992a) describes an abstract machine for the system. The system is also usable as a general-purpose logic programming language (since CLP($\Re$) subsumes Prolog).

A clp($\Re$) program is a collection of rules in the sense of Prolog but where the body can contain constraints and the terms are defined more generally. Basically, constraints are equations or inequalities in the form

$$Expression_1 \bullet Expression_2, \qquad \text{with } \bullet \in \{=, >, \geq, <, \leq\}$$

where the expressions to both sides of the constraint are built from real constants, variables, real terms constructed with unary negation and more complex terms constructed from the usual arithmetic operators such as $+, -, *$ and $/$. For instance,

$$X + 3.0 * Y \leq Z \qquad \text{and} \qquad 3.451 + W = 24 * Y$$

are examples of constraints in CLP($\Re$).

This system was extended to allow for powerful facilities for meta programming with constraints (Heintze et al., 1989). Moreover, CLP($\Re$) has been proved to be very useful on practical applications and have been published in diverse areas and much has been written on them e.g., molecular biology (Yap, 1991), testing of protocols (Gorlick et al., 1990), electrical engineering (Heintze et al., 1992), model-based diagnosis of analog circuits (Biasizzo and Novak, 1995) and option trading (Huynh and Lassez, 1988) among others.

The success of the CLP($\Re$) system lead to a number of CLP systems managing arithmetic constraints in a similar form. For instance, clp(Q,R) (Holzbaur, 1995) is

another CLP language that allows the solving of linear equations over rational or real valued variables, covers the lazy treatment of non-linear equations, features a decision algorithm for linear inequalities that detects implied equations, removes redundancies, performs projections (quantifier elimination), allows for linear dis-equations and provides for linear optimisation. This system is nowadays part of the distribution of the most important Prolog systems such as SICStus Prolog (Carlsson et al., 1997), ECL^iPS^e (Aggoun et al., 1995) and CIAO Prolog (Hermenegildo et al., 2000).

Also CHIP, the pioneering CLP system on FD, allowed a kind of constraint solving on *rational terms*. Basically a rational term is a term constructed from rational values and variables and the operations + and *. CHIP solved linear equations, inequalities and dis-equations through a symbolic Simplex-like algorithm. Therefore, applications of the CHIP system in the rational term domain are problems involving linear programming.

Solving non-linear constraints over real numbers is a complex problem that was not really solved in CLP($\Re$) and its resolution was delayed until these constraints became linear. This efficient implementation method has the disadvantage that sometimes computed answers are unsatisfiable or infinite loops occur due to the unsatisfiability of delayed non-linear constraints. For this reason, from the original approach of the CLP($\Re$) system, other CLP approaches try to improve the resolution of non-linear constraints.

(Hong, 1993) reported about their experience in combining CLP with two algebraic methods, the method of *partial cylindrical decomposition* (Arnon et al., 1984) and the method of *Gröbner bases* (Buchberger, 1997), for solving non-linear constraints over real numbers. These methods were developed in computer algebra during last four decades. The prototype implementation RISC-CLP(Real) demonstrated that non-linear constraints deserved more attention.

Also, (Hanus, 1993) proposed to solve the delaying problem of non-linear constraints by using a more powerful constraint solver which can deal with non-linear constraints like it is done in the language RISC-CLP(Real). The proposal is to get a compromise between these two extremes by characterizing a class of CLP($\Re$) programs for which all delayed non-linear constraints become linear at run time. Programs belonging to this class can be safely executed with the efficient CLP($\Re$) method while the remaining programs need a more powerful constraint solver.

Afterwards, the system QUAD-CLP($\Re$) (Pesant and Boyer, 1994) was built on top of the CLP($\Re$) system to provide a further treatment of non-linear arithmetic constraints over the reals as opposed to delaying them unconditionally. It concentrates on quadratic constraints, rewriting them in such a way they can actually be decided upon either generating a conservative approximation of them (while still delaying them) or potentially improving control over the computation. In both cases the idea is to fall back on linear constraints, more easily handled. QUAD-CLP($\Re$) is an incomplete solver for non-linear constraints catering for problems of respectable size which require a certain amount of reasoning on these constraints but cannot afford the prohibitive cost of a complete treatment.

These works lead to present the solving of non-linear constraints like a very active

field inside the CLP community.

Constraint solving on the real domain is becoming an important issue in the context of functional logic languages. In (Arenas et al., 1996), a declarative language, called CFLP($\Re$), that integrates lazy functional programming, logic programming and constraint solving over real numbers is proposed. The execution mechanism of the language consists of a combination of lazy narrowing and constraint solving. The main drawback of this language is in its implementation. The method for implementing the language consists of the translation of CFLP($\Re$) programs into a logic programming language supporting real arithmetic constraint solving.

2.6.3 Sets

Finite set is other important domain employed traditionally in LP. Sets enable the modelling of problems formulated on combinatorial solving and on natural language processing. A lot of real problems involves sets or multisets, relations or graphs that can be tackled by means of sets.

The first attempt to cover with sets in CLP was done in (Walinsky, 1989) that proposed to define regular sets over *words* as a new domain in the CLP language CLP(Σ^*). Regular sets are finite sets composed from finite strings and the constraints in CLP(Σ^*) have the form

$$A \; in \; (X.\text{“}ab.\text{”}Y)$$

which associates the variable A to any string containing the substring “*ab.*”. Regular sets cannot be viewed in the general concept of sets but CLP(Σ^*) was the first initiative to tackle with sets in the CLP framework.

Afterwards, CLPS (Legeard and Legros, 1991) is another attempt founded in the notion of sets of finite depth over Herbrand terms (e.g., a simple set $\{1, 2, 3, 4\}$ is of depth 1, a set $\{\{1, 2\}, 3, 4\}$ is of depth 2, a set $\{\{\{1, 2\}\}, 3, 4\}$ is of depth 3, and so on). The satisfaction of constraints is performed by checking the consistency over set elements as domain variables.

One of the most influential works on CLP(Sets) was described in (Gervet, 1994) where a CLP language, called *Conjunto*, to manage set interval constraints is defined. The motivation was a desire to combine the efficient constraint satisfaction techniques with the declarativity of Prolog in order to solve combinatorial problems based on sets, relations or graphs (e.g., set partitioning, set packing, maximum-minimum set cover, etc.). On this line, (Gervet, 1997) proposed to have a deterministic set unification procedure consisting of testing in polynomial time the equality between set variables and ground sets. To cope with it, a set domain was approximated by a set interval specified by its upper and lower bounds, thus guaranteeing that a partial order exists. A set variable s is then associated to a set interval in the form

$$s \in [l, u] \qquad \text{with } l \subseteq u$$

and l and u are integer sets e.g., $s \in [\{1\}, \{1, 3, 5, 7\}]$. This approach enables an important improvement with respect to previous works: it provides the possibility to use consistency techniques to reason in terms of interval variations.

Recently, (Kozen., 1994; Kozen., 1998) has proposed the use of set constraints to define a CLP language over sets of ground terms that generalizes ordinary LP over an Herbrand domain. Also, (Dovier et al., 1996) develops the language $\{log\}$ that provides the basis to embed set constraints in the form $\{x\} \cup Set$ in a logical language. The satisfiability of constraints is based on a non-deterministic selection of constraints by taking into account all the possible substitutions between the elements of two sets. Unfortunately, this leads to a hidden exponential growth in the search tree.

Currently, the set domain is included in a number of main CLP systems such as ECL^iPS^e (Aggoun et al., 1995) or Oz (Müller and Müller, 1997). For an extended overview of sets in (C)LP see (Gervet, 1997; Stolzenburg, 1996).

2.6.4 (Pseudo-)Booleans

Among the usual domains studied in CLP, Booleans are widely studied because of its usefulness since many practical problems involve Boolean variables (also called *0-1 variables*). The study of Boolean problems comes (Williams, 1993a) from the modelling with Boolean variables of problems in automated theorem proving, circuit diagnosis and verification, artificial intelligence, etc. Thus, it is not surprising that this domain is included in existing CLP languages, sometimes with a dedicated solver and other times by providing libraries to manage Boolean constraints.

A Boolean constraint solver usually provides support to solve, at least, the following Boolean formulas

$$X \vee Y \equiv Z,$$
$$X \wedge Y \equiv Z,$$
$$X = Y \text{ and}$$
$$X \equiv \neg Y$$

where variables X, Y and Z are Boolean variables i.e., those variables taking values in the Boolean domain. It is desirable that it also incorporates other Boolean formulas such as the *exclusive or*, the *not and*, the *not or*, the *equivalence* or the *implication*. However, with the first four formulas is possible to define the rest.

The basic processing of a set of Boolean constraints is to study the satisfiability of the Boolean formulas.

It is worth clarifying the distinction between stand-alone Boolean solvers, that is, those solvers dedicated exclusively to the solving of Boolean constraints, and those CLP languages that provide either a constraint solver as a library module or the capacity to use Boolean constraints and solve them. In this last category we can find the CHIP language (Büttner and Simonis, 1997) and Prolog III (Benhamou, 1993) that offer special purpose Boolean algorithms in which the processing of Boolean constraints is done in the unification step. The only necessary constraint is the equality over Boolean terms. In these languages, a Boolean term is constructed from constants, Boolean values, Boolean variables and the logical operators *and, or, not, xor, nand*

and *nor*. The algorithm for unification (i.e., to compute the most general unifier of two Boolean terms) is based on variable elimination.

Other specialised algorithms for Boolean constraint solving should be mentioned. For instance (Sato and Aiba, 1993b) proposes the Boolean solver algorithm for the CAL language (Aiba et al., 1988; Aiba and Sakai, 1989) (in fact it is an application of the Buchberger algorithm (Buchberger, 1987a) to Boolean rings). The CAL language provides a Boolean algebra with symbolic values, where equality between Boolean formulas expresses equivalence in the algebra (see Section 2.8 for more information about the CAL language). Also other full languages such as GNU Prolog (Diaz and Codognet, 2000), Prolog IV (N'Dong, 1997), IF/Prolog (If/Prolog, 1994) SICStus (Sicstus manual, 1994) and B-Prolog (Zhou, 1997) incorporate library modules to manage Boolean constraints. The CHR language (see Section 2.4.2) deserves specific mention since, due to its flexibility, the Boolean solver can be completely defined at the user level.

The idea of considering Booleans as the subset $\{0, 1\}$ of integers, that is, as a particular case of FD was first introduced in the CHIP language. This idea allows for a generalisation of the Boolean formulas as for instance the idea of reified constraints (see Section 2.7). This approach of CHIP was so successful that it became the standard tool in the commercial version of CHIP whereas its specialised Boolean solver could be chosen as an option. The primitive Boolean constraints of CHIP were coded internally in a black box approach and, thus, wired inside the internal Boolean solver. Afterwards, the idea was extended to the clp(FD/B) system (Codognet and Diaz, 1993). The extension consisted of the integration of the Boolean solver in the specialised solver for FD of the clp(FD) system. Boolean constraints such as *and*, *or* and *not* were decomposed in simplifier X *in* r expressions for the finite domain $\{0, 1\}$. The schema of propagation was very simple and supposed to open the black box approach of the Boolean solver of CHIP to a glass box approach. Also, in (Codognet and Diaz, 1996a) it was shown how the clp(FD/B) efficiency was on average an order of magnitude faster than the CHIP Boolean solver. Moreover, this glass box approach was more efficient, surprisingly, than some special-purpose Boolean solvers.

The clp(FD/B) was specialised exclusively for the Boolean domain in the system clp(B) (Codognet and Diaz, 1994). This system introduces specific optimisations for the Boolean domain and removes, from the clp(FD/B) system, data structures that were only used in the FD and were useless for the Boolean domain. The resulting system was a simple and compact Boolean solver, more efficient than the previous system and based on the glass box approach of indexicals (see Section 2.4.1) but with a specialised X *in* r constraint primitive for the Boolean domain.

Recently a generalisation of the Boolean constraints, called the *pseudo-Boolean* constraints, is being studied. Pseudo-Boolean constraints are equations or inequalities between multilinear integer polynomials in 0-1 variables (i.e., variables where 0 denotes *false* and 1 denotes *true*) (Barth, 1996; Bockmayr, 1994). Pseudo-Boolean constraints are thus a restricted form of the FD constraints.

EXAMPLE 2.11 *To model the interaction of n objects $ob_1, \ldots, ob_n$, each of which can be chosen or not, it is natural to use a quadratic pseudo-Boolean function of the*

form

$$\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}X_iX_j.$$

where a_{ij} measures the interaction between objects ob_i and ob_j and the 0-1 decision variables X_i indicates whether ob_i is chosen or not (Barth and Bockmayr, 1996). As most of the existing solvers cannot deal with non-linear pseudo-Boolean constraints, the usual technique is to linearise them and then solve them in a linear 0-1 constraint solver.

There have been several CLP languages developed for the pseudo-Boolean constraints. For instance, the CLP language clp($\mathcal{PB}$) allows to model and reason about 0-1 problems. It was introduced in (Bockmayr, 1993) and a prototype implementation is available in (Barth, 1994). Given a set of 0-1 Boolean problems, the solver computes an equivalent set of simplifier clauses that are provided to a 0-1 constraint solver for their solving. Pseudo-Boolean constraints correspond to non-linear 0-1 programming problems (Hansen et al., 1993) when they are used on operation research.

2.6.5 Interval Constraint Arithmetic: CLP(Intervals)

The real domain is a continuous domain and algorithms to solve constraints defined on the real domain are studied in an idealised setting. However, in practice, these algorithms are accepted to be no longer valid since real numbers are approximated by floating point numbers and this finite representation of real numbers in a computer prevents them from accurately solving real constraints. This means that there will be some approximation errors. In fact, the methods of conventional numerical computation involve floating point approximations to a finite set of real values, that is to say, basically find approximate solutions (inside a error bound) to a set of constraints.

An alternative approach are the methods using interval arithmetic that compute a solution as a union of intervals such that the mathematical solution relies in one of them. Only values not belonging to the solution are removed and thus it is possible to guarantee that the real values of the solution are in the interval returned as the solution. For interval methods one has to find the right abstraction of floating point numbers in terms of real numbers.

The first publication about interval arithmetic is assigned to Moore (Moore, 1966). Moore replaces any real constant by an interval containing it and extends real operations to intervals in such a way that the rounding error is bounded in each operation. If the merit of interval arithmetic is attributed to Moore, the beginning of interval constraints can be associated to Waltz and his paper (Waltz, 1975) in which constraints were propagated to reduce sets of possible values. The next development of constraint interval arithmetic was its application to reals (Davis, 1987).

The merit of extending the LP paradigm to include interval arithmetic is due to (Cleary, 1987) that extends the typical approach of interval arithmetic in the functional setting to the relational theory. Independently (Hyvönen, 1989) also discovered interval

constraints and the application of interval arithmetic to constraint solving. The basic interval constraint methods are then enhanced by incorporating Newton's method (Benhamou et al., 1994; Van Hentenryck et al., 1997). More recently Hickey (Hickey et al., 1999; Hickey, 2000) continue the advance in the search of a unified framework for interval constraints nd interval arithmetic.

In general, the basic idea under the CLP(Interval) schema is to evaluate every numerical expression by using intervals instead of floating point numbers with the aim of not loosing numerical accuracy. CLP(Interval) has been shown to be a very powerful scheme for solving non-linear constraints.

From the implementation point of view, the CLP(Interval) approach is to build into the language itself a general interval-based constraint solver that guarantees the completeness of the solutions (i.e., all solutions in the input are retained). CHIP (Lee and van Emden, 1993; Van Hentenryck, 1988) was the pioneering system to show that the idea of the interval constraints was widely applicable. Afterwards, the system BNR-Prolog (Older and Vellino, 1993) was built mixing the ideas of Davis and Cleary on top of a logical language (i.e., Prolog). In general, it is possible to say that, historically, there have been two approaches to implement interval arithmetic constraint solvers. One approach, the *hull consistency* approach, is represented by the system CLP(BNR) (CLP(BNR), 1988; Older and Benhamou, 1993; Older and Vellino, 1993; Benhamou and Older, 1997) and the other approach, the *box consistency* approach, is represented by the systems Newton (Benhamou et al., 1994; Van Hentenryck et al., 1988) and Numerica (Van Hentenryck et al., 1997; Van Hentenryck, 1998).

- The *Hull consistency* approach.

  This approach consists of translating complex arithmetic constraints in primitive constraints and then performs a constraint contraction for each of the primitive constraints. The mechanism of propagation is as usual, that is, constraints shared variables so that contraction usually has to be performed multiple times on any given constraint: every time another constraint causes the interval for a variable to contract, all constraints containing that variable have to have their contraction operators applied again. Often, the termination of this constraint propagation is guaranteed in the fact that reals are floating point numbers and, thus, there will be a finite number of contractions. That is, a finite number of contractions suffices to reach a state where all constraints yield a null contraction. A constraint propagation algorithm terminates when this case is found.

  The main drawback of this approach is that the decomposition of complex variables introduces new variables that lead to unnecessary approximations.

  CLP(BNR) is a Prolog-based language that incorporates an arc consistency algorithm on interval-bounded constraints which handles general algebraic constraints over continuous, integer and Boolean variables. This allows programmers to express systems of non-linear equations on real intervals that can be arbitrarily mixed with integer and Boolean constraint equations. In CLP(BNR) each constraint is decomposed into primitive constraints, and then a general constraint

solving engine is invoked to repeatedly contract each primitive constraint until some termination condition is satisfied.

- The *Box consistency* approach.

  Systems based on this approach implement the constraint-solving algorithm as a combination of traditional numerical methods such as interval and local methods and constraint satisfaction techniques. Box consistency allows to efficiently process complex constraints without decomposition.

  Two main exponents of this approach are the systems Newton and Numerica that solve, using interval reasoning, systems of non-linear equations and inequalities as well as problems of optimisation.

Recently there have appeared some languages that combine both approaches such as DecLIC (Goualard et al., 1999).

The interested reader is referred to standard references in interval constraints such as (Older, 1989; Benhamou and Older, 1997; Van Emdem, 1997). For an overview of the applications of the interval arithmetic in a relational setting see (Majumdar, 1997). Another standard references are (Alefeld and Herzberger, 1983) and (Hansen, 1992).

Non-Usual Interval Constraints in CLP

Traditionally, interval constraints have been applied to the real domain by approximation methods in which a real number is usually approximated by an interval containing floating point numbers. The floating point number domain is effectively finite so that constraint propagation is executed in a finite domain (which guarantees some properties as termination). However, as already shown in Sections 2.4.1 and 2.6.3, interval constraints have also been used in other domains such as the finite domain and the set domain.

2.6.6 CLP(Trees)

Trees allow the modelling of problems that other domains cannot model. They can be traversed and processed in different ways and they show a high potentiality in the constraint solving. Moreover, on order-trees, they can also be kept ordered so that search is relatively cheap. For these reasons, CLP has been also applied to the Prolog-like tree domain.

The Herbrand domain is the only domain for LP and thus it is present in all the CLP languages. In fact, usual LP is often regarded as CLP over the Herbrand domain, where Herbrand terms are a representation of *finite trees* and the constraints are restricted by the equality. Some existing systems can deal specially with constraints over Herbrand's universe. For instance the system HAL (Demoen et al., 1999a; Demoen et al., 1999b) is a new constraint logic programming language specifically designed to support the construction of and experimentation with constraint solvers. HAL programs provide tests for equality and construction and deconstruction of ground terms.

The constraint system FT (Aït-kaci et al., 1994) also provides a universal data structure based on trees. It presents an alternative to the Herbrand constraints over constructor trees. The constructors in FT are more general than those of Herbrand, and the constraints of FT are of finer grain and of different expressiveness. The essential novelty of FT is provided by functional attributes called features which allow representing data as extensible records, a more flexible way than that offered by Herbrand's fixed arity constructors.

LIFE (Aït-kaci and Podelski, 1993) is an experimental language proposing to integrate logic programming, functional programming and object-oriented programming. This language enables the computation over an order-sorted domain of feature trees by allowing the equality (i.e., unification) and entailment (i.e., matching) constraints over order-sorted feature terms. It is the precursor of other languages such as LOGIN (Aït-kaci and Nasr, 1986) and Le Fun (Aït-kaci et al., 1987).

A number of modern LP languages such as Prolog III (Colmerauer, 1990) and SICStus (Sicstus manual, 1994) offer a computation domain based on *rational trees*. The importance of this domain is demonstrated by the fact that Prolog III and SICStus use rational tree unification as the default solver (King, 2000). A rational tree is a tree with a possibly infinite number of nodes but where the number of branches emanating from each node is finite. The use of such trees allows faster unification (due to the omission of the occurs-check) and an increase of the declarativity. Unfortunately the use of rational trees also involves a surprising number of problems (Bagnara et al., 2001). For instance, many of the built-in and library predicates are ill-defined for such trees and need to be supplemented by run-time checks whose cost may be significant. Also observe that the domain of finite trees is the intended computation domain of most LP languages and thus some widely-used program manipulation techniques assume this computation domain. This means that a number of these techniques either are not applicable to infinite trees or have not been proved to be correct in the rational tree domain.

2.7 Specialised Constraints

Specialised constraints are a kind of constraints that are built into the system to provide specific advantages. In general, they are black box constraints since the user knows what these constraints do and how to use them, but the user is far from knowing the details about how these constraints work operationally.

Specialised constraints can be grouped into two classes: the first class contains those constraints whose aim is clearly to add expressiveness to the formulation of a CSP. The constraints into this class can be used in different settings and their formulation depends on the ability of the programmer. Into this class we find the *reified constraints* and the *meta-constraints*.

Reified Constraints (RCs) reflect the validity of a constraint into a Boolean variable. Constraints in *reified* form allow their fulfillment to be reflected back into an FD

variable. In general, a RC is an expression of the form

$$b \equiv c$$

where b is constrained to *true* as soon as c is known to be true and to *false* as soon as c is known to be false. On the other hand, constraining b to *true* imposes the constraint c, and constraining b to *false* imposes its negation.

EXAMPLE 2.12 *The constraint*

$$x \equiv (y + z > v)$$

constrains x to true as soon as the inequation is known to be true and to false as soon as the inequation is known to be false. On the other hand, constraining x to true imposes the inequation, and constraining x to false imposes its negation.

RCs are both useful and difficult to implement and normally provided as black boxes. As written in (Marriot and Stuckey, 1998, Page 284): "RCs are somewhat complex to implement since they require the solver to determine whether or not a constraint is implied by the current constraint store and whether or not its negation is implied. However because of the usefulness of reified constraints, some finite domain constraints solvers provide them".

Meta-Constraints provide a means of expressing constraints over constraints. A form to do this is by using logical connectives that have to be applied directly to a constraint or to an expression using these logical connectives.

EXAMPLE 2.13 *The following logical formula*

$$(x < 4) \equiv (y < 3) \wedge (z \geq 8) \vee (w \leq 3) \wedge (w = y)$$

constrains x to be smaller than 4 when $y < 3$ and $z \geq 8$ or when $w \leq 3$ and $w = y$. Also, as soon as x is known to be smaller than 4, the disjunction of constraints in the right side of the formula is imposed.

The second class of complex constraints contains those constraints that are implemented for specific applications with the objective of leading to a better performance. In general these constraints allow a more concise modelling of a problem although their main drawback is that their use is usually restricted to the setting of the application for which they were implemented. These constraints give better propagation than that provided by an equivalent conjunction of simpler primitive constraints and lead to a more efficient program since they wrap very specialised algorithms for propagation. For instance, some constraints falling into this class are the *element* constraint, the *cardinality* constraint, the *all different* constraint and the *cumulative* constraint.

- The *cardinality* operator (Van Hentenryck and Deville, 1991) is used to express any Boolean combination of constraints.

- The *all different* constraint (van Hoeve, 2001) states that all variables in this constraint must be pairwise.
- The *element* constraint (Dincbas et al., 1988a) checks if an element belongs to a list and allows to define constraints on the element, the list or even on the index of the element in the list.
- The *cumulative* constraint (Aggoun and Beldiceanu, 1992) that is a specialised and useful constraint to solve scheduling and placements problems.

Complex constraints are generally built into the language and, in general, the consistency methods for these constraints are very complicated since they have into account numerous possibilities for the interaction of constraints depending on the state of the variables as for example which of them are instantiated.

2.8 Other CLP Languages

In this section we cite some languages that have contributed in certain form to the development of CLP and that were not cited previously. Of course the list is not exhaustive.

CAL (Aiba et al., 1988; Aiba and Sakai, 1989) computes over real numbers and over a Boolean algebra with symbolic values. In the first domain, constraints are equations between polynomials and in the second one the equality between Boolean formulas expresses equivalence in the algebra. CAL is able to handle non-linear polynomial equations on complex numbers by employing the Buchberger algorithm (Buchberger, 1987b; Buchberger, 1987a) as one of its main solvers. This algorithm calculates a canonical form of systems of equations called Gröbner bases and has been studied, during years in computer algebra, and applied to different fields (e.g., handling robot kinetics or computational geometry).

Gröbner bases have also been applied in other languages to solve non-linear constraints such as **CoSAc** (Monfroy et al., 1995) that is a CLP system for solving non-linear constraints based on cooperating solvers. Its non-linear constraint solver is based on GB (Faugere, 1994), the fastest known system for Gröbner bases. It is constructed as a architecture of clients/servers in which each server is specialised for a particular computation.

AKL (Carlson et al., 1994b) (previously Andorra Kernel Language (Janson and Haridi, 1991), now Agent Kernel Language) is a concurrent constraint programming language that supports both Prolog-style programming and committed choice programming. Its control of don't-know non-determinism is based on the Andorra model, which has been generalised to also deal with non-determinism encapsulated in guards in a concurrent setting.

TOY (Caballero et al., 1997) is a system for functional logic programming supporting lazy narrowing, higher order (HO) features (including HO logic variables), disequality constraints (for constructed data terms) and non-deterministic functions.

TOY supports term (strict) equality and disequality constraint solving. Toy also allows the ability to handle linear constraints over real numbers although this is done via the SICStus system.

Recently, (Van Hentenryck et al., 1999) describes the modeling language called OPL (*Optimization Programming Language*) to combine high level algebraic and set notations from modeling languages with a rich constraint language and the possibility to specify procedures and search strategies. This language is described in detail in (Van Hentenryck, 1999).

2.9 Constraints on Other Paradigms

The real success of CP languages motivated a movement of the constraint paradigm to another paradigms. The early concepts of CLP were then adjusted to better serve in different areas of applications. In this section we cite some areas, but of course, the list is not exhaustive.

Concurrent constraint programming (CCP) is based on the asynchronous communication between "agents" by using constraint entailment. A user-defined constraint is viewed as a *process* and a state is regarded as a network of processes linked through shared variables by means of the store. Processes communicate by adding constraints to the store and synchronise by waiting for the store to enable a delay condition.

In (Maher, 1987), concurrent logic languages (Shapiro, 1989) were generalised to the constraint setting by recognising that the synchronisation operator used in concurrent logic languages can be thought as constraint entailment. Then, in 1989, (Saraswat, 1989) described an elegant theoretical model for concurrent constraint languages and provided the term of "concurrent constraint programming". Since then there has been considerable progress on different aspects of CCP (Saraswat, 1993; Saraswat, 1992).

In particular, concurrent constraint logic (CCL) languages allows the interaction of processes which may interact one with each other. The communication and synchronisation are done by asserting and testing of constraints. The most important CCL languages are mainly based on two delay conditions called *ask* and *tell*. These conditions were defined in (Saraswat, 1988; Saraswat, 1989). An *ask* delay condition is of the form $ask(C)$ and is enabled when the constraint store implies the constraint C. For instance *emptylist(Y)* is equivalent to $ask(Y = [])$. Moreover, ask conditions can involve local variables by means of existential quantifiers. For example, the condition *nonemptylist(Y)* is equivalent to $ask(\exists X \exists L.Y = [X \mid L])$ since this is enabled wherever there exists values for X and L such that the constraint store implies $Y = [X \mid L]$. Another delay condition is the *tell* condition that has the form $tell(C)$ and is enabled if constraint C is consistent with the constraint store.

The major difference between CCP and CLP languages is in how multiple rules defining the same predicate are handled. In CLP languages, each rule is tried until an answer is found i.e., they employed don't know determinism. In CCP languages the evaluation mechanism delays choosing which rule to use until at least one of the guards

is enabled (a guard consists typically of a *tell* and *ask* condition). If there are more than one rule to choose then one is arbitrarily chosen. Independent of what happens in the future, backtracking is never done so that is responsibility of the programmer to provide each rule with a guard that ensures once it is enabled. Thus CCP languages provide *don't care non-determinism* (i.e., if more that one guard is enabled we do not care which of the corresponding rules are used since any will be correct).

(de Boer and Palamidessi, 1994) provides a complete survey that shows the motivations to extend the concurrent setting to the (C)LP paradigm and analyses the CCP paradigm and the main approaches to the semantics foundations.

Constraint functional programming raised from the integration of constraints in the functional paradigm. One way to do it is to embed the CLP approach into the functional setting. This is natural since the functional paradigm is not too far from the logic paradigm since both are declarative in the sense that the programmer just specifies what to do but not how to do it. From several years ago, there have been a growing interest in the combination of these paradigms (FLOPS, 2001; JFLP, 2000) and this has been an active area of research. One way to get this integration is by viewing constraints as functions embedded in a functional language and returning a list of answers. The cardinality of this list can be non-finite so that a lazy functional language (Hughes, 1989) is required as the underlying language.

The dual approach means to embed functions into a CLP language. This seems an ideal situation since n-ary functions can be viewed as predicates with an arity of n+1 where an extra argument is added to get the result of the evaluation of the function. However, several problems remain with respect to the evaluation mechanism of constraint solving over user-defined functions.

Another approach consists of extending the lambda calculus (Barendregt, 1984; Barendregt, 1990) (i.e., the rewrite system providing the evaluation mechanism for functional languages) by including a global constraint store. Constraints are sent to this store by function application. The problem now is how this store can have an active role in the program evaluation. (Crossley et al., 1996) described a method called *constrained lambda calculus*, in which only definite values can be communicated from the store. The store is used to determine the value of the variables so that if the value of a variable is determined then it replaces the variable by its value throughout the lambda expression. The problem is that the store plays a very passive role in this process (specially with respect to the search since it cannot guide it).

Recently a complementary approach has born from integrating constraints into the so called functional logic languages that are programming languages that combines the characteristics of the logical and functional paradigms. So far, real constraints have been integrated, with major or minor success, in these languages (Lux, 2001; Arenas et al., 1996). An active research is currently done in the integration of other constraints in this kind of languages (Arenas et al., 1994; Arenas et al., 1999; Cazorla, 2001).

Imperative constraint programming embeds constraints into the imperative setting. In particular, some object oriented languages allows the programmer to specify constraints and thus to formulate, in a concise way, the relation they want among several variables. This integration of constraints into the imperative setting guarantees the efficiency of the program (Freeman-Benson and Borning, 1992). Also, other object oriented languages provide incremental constraint satisfaction similar to that in the CLP languages. This means that constraints are embedded into a host imperative language by means of specific built-in solvers (Ilog, 1995). Instead of the existence of imperative languages that provide constraint solving to the user, it remains yet a lot of research to do in the integration of constraints with imperative languages.

Multiparadigm constraint programming means to combine CP with several paradigms in one setting. This is the idea presented in (Smolka, 1995) and implemented in the language *Oz* (i.e., Mozart). This language combines the characteristics of CLP languages, functional languages and concurrent languages. Search is implemented quite different from the CLP languages since search is programmable. Instead of the typical LP approach of left-right first-depth, search strategies in *Oz* are encoded in *search procedures* to explore the search space. Moreover, computation can be suspended in the choices until a search procedure is explicitly provided. *Oz* generalises the CLP and CCP paradigms and provide a very flexible approach to constraint programming.

2.10 Concluding Remarks

This chapter provides a general overview of the state of the art in current C(L)P systems. The emphasis has been on the different solving mechanisms for each instance of the CLP scheme. We have shown how the mechanisms may be divided into two main categories: transparent (the glass box approach) and opaque (the black box approach). Both categories have been discussed and their main advantages and disadvantages presented.

We have tried to emphasize the multi-disciplinary character of the CP paradigm that covers aspects as diverse fields in mathematics, computer science, AI, programming languages, symbolic computing and computational logic among others.

In the rest of this thesis, we continue to present the state of the art in CLP by discussing related work with specific issues described in this document (e.g., issues such as frameworks for generic constraint propagation, techniques for constraint branching, mechanisms of solver cooperation and different techniques for the implementation of solvers).

Part II

Comparative Framework

Chapter 3

A Comparison of Glass Box Systems

Your true value depends entirely on what you are compared with.

Bob Wells

3.1 Introduction and Motivations

Evidence of the success of the CLP paradigm (Csontó and Paralič, 1997; Jaffar and Lassez, 1987) can be found in the increasing number of CLP systems now being used for many real-life applications (PAPPACT'98, 1998). As declared in Part I, there are two main reasons for this success: first, CLP extends the logic programming paradigm enabling more declarative and readable solutions and, secondly, it supports the propagation of constraints for specific domains, providing an efficient implementation for the computationally expensive procedures. However, CLP systems differ significantly both in how solutions may be expressed and the efficiency of their execution. It is important that both these factors are taken into account when choosing the best CLP system for a particular application. In fact, a wrong choice for an application may be disastrous, not only relative to its efficient performance, but also with respect to the code clarity of the solution, which is important for future modifications. In spite of this, there appears to be no impartial set of guidelines for choosing an appropriate constraint system for solving a specific constraint satisfaction problem.

Of the domains for CLP, the finite domain (FD) (Van Hentenryck, 1989) is one of the best studied since it is a suitable framework for solving discrete constraint satisfaction problems. FD is particularly useful for modeling problems such as scheduling, planning, packing, timetabling and as a consequence most of the CLP systems provide substantial FD libraries. Also, as already pointed out in Chapter 1, we are interested in the glass box approach. (In Section 2.3.6 we show the advantages of this approach with respect to the classical black box approach.) We find that most of the existing

glass box systems are defined for the Finite Domain. For this reason, we consider, in our comparison of systems only the FD and the Boolean domain (which is sometimes regarded as an instance of the FD -see Section 2.6.4). The main aim of our comparison is to the choose an adequate glass box approach to implement our idea of generic solver. The chosen approach has to allow certain flexibility in the formulation of problems as well as to show an acceptable performance of the systems over which is implemented.

Our study also provides an impartial comparison of a number of FD solvers for CLP. We contrast eight constraint systems; ECL^{i}PS^e (Aggoun et al., 1995), Oz (Smolka, 1995), Ilog SOLVER (Ilog, 1995), clp(FD) (Codognet and Diaz, 1996a), CHR (Frühwirth, 1998), SICStus (Sicstus manual, 1994), IF/Prolog (If/Prolog, 1994) and B-Prolog (Zhou, 1997). We chose these particular systems since they cover the main kinds of FD solvers and are all very popular within the CLP community. Because of our limited resources, other interesting CLP systems such as CHIP (Van Hentenryck, 1988) and Prolog IV (N'Dong, 1997) are not included in this comparison.

Although the constraint systems have been tested on the solving of a number of traditional benchmarks, most comparative work has been done by the language implementers themselves (Carlsson et al., 1997; Codognet and Diaz, 1994; Codognet and Diaz, 1996a; Cras, 1993; Müller and Würtz, 1996; Puget and Leconte, 1995; Sidebottom, 1993). Furthermore, these benchmarks have been used in the development as well as the assessment of the languages so that such tests are biased. Here we are not a designer of any of the CLP languages studied here so that the comparison is more impartial than previous ones.

We have chosen a particular kind of logical puzzle, called Self-Referential Quiz (SRQ) as the original benchmark for the comparison. This is one of a new class of puzzles first described in (Henz, 1996) for demonstrating the meta-reasoning capabilities of the Oz FD system. Because of the self referential nature of the problem, it is particularly suitable for examining the ease with which the different languages can be used for applications requiring meta-reasoning. Thus, here we use the SRQ to illustrate how each of the languages support reification in the Boolean domain and meta-constraints in the FD. Also, we use the SRQ for the efficiency comparisons since this is a non-standard benchmark not used in the development of any of the systems (apart from Oz). Hence these particular comparisons are essentially fair and unaffected by any bias the implementation may have to perform well on the standard benchmarks.

As it was shown in Section 2.2.3 and also is discussed in Section 3.5.1, constraint solving can be viewed as a combination of two processes, constraint propagation and labeling. Since constraint propagation is the main reason for choosing the CLP technology, we concentrate on comparing the constraint propagation methods. However, to ensure fairness in the efficiency testing, we consider two distinct labelings, *naive* and *first fail*.

It is well known that a single benchmark is not adequate for evaluating a programming system. Moreover, SRQs can be solved quickly and, indeed, are not easily scalable. Thus, to make the comparison fair and thus more objective, we extended our study to other problems (two of which are scalable) comparing mainly the efficiency of the solutions. We chose some well-known problems and used, wherever available,

solutions provided with the systems or, if not, directly from the implementers.

3.1.1 Chapter Structure

The rest of the chapter is organised as follows: Section 3.2 describes the essential features of the FD constraint systems for the eight languages considered in this chapter. In Section 3.3, the SRQ puzzle is defined and two different solutions to it are described. Section 3.4 shows, in a tutorial way, how each language can be used to express reified constraints and meta-constraints. Extracts from the SRQ formulations in each of the languages are used to highlight the main differences between them. In Section 3.5, a comprehensive efficiency comparison is reported and the results are discussed. The chapter concludes with a discussion about related work, a summary of the main results and a listing of the major contributions of the chapter.

3.2 The Constraint Systems Tested

In this section, we describe briefly the constraint systems considered in the chapter. All of them are *glass box systems* but we classify them as *glass box* or *black box* depending on the level of knowledge affecting to the user. So, first we explain what we mean by these classification. Within each classification, we then describe the constraint systems compared. We conclude the section with a short note discussing the ease with which the languages could be learned.

3.2.1 Classification of the Systems

In Section 2.3.6, the glass box and black box approaches were discussed and it was shown that the distinction between a *glass* or a *black* box language is not always clear. All the systems studied in this chapter are glass box systems in a strict definition since all of them allow the user to define new constraints. However, some of them require a detailed knowledge of their implementation in order to specify the propagation mechanism of the new constraints. For this reason, in the following, we classify these systems as black box systems.

3.2.2 *Glass Box* Languages

In Section 2.4, we showed that there are two main different kinds of *glass box* languages. These differ in the way that constraint propagation may be defined: either using a single form of relational construct called an *indexical* (Codognet and Diaz, 1996b) or by means of special *Constraint Handling Rules* (CHRs) (Frühwirth, 1998).

Indexical languages

The indexical approach was introduced and explained in Section 2.4.1. The three languages examined here that support indexicals are clp(FD) 2.21, SICStus 3#5 and IF/Prolog 5.0.

A Language with Constraint Handling Rules

CHRs were introduced and explained in Section 2.4.2. For a CHR language, we use the library that is built on top of ECL^{i}PS^e 3.5.2, amalgamating the CHRs with the underlying language.

3.2.3 *Black Box* Languages.

The four *black box* languages examined here were described in Section 2.5 and are:

- Oz 2.0.
- ECL^{i}PS^e 3.5.2.
- Ilog SOLVER 3.1
- B-Prolog 2.1

3.2.4 Ease of Learning

We now discuss the ease with which we could learn the different CLP languages based on our experience of writing solutions to the SRQ problem described in Section 3.3. The ease which a new language may be learned depends on many factors, including the learners background, availability of helpful documentation and personal tuition. From our perspective (used to both Prolog and functional programming languages), we found clp(FD), SICStus and IF/Prolog the simplest to master since they are based on one main constraint and this was in the form of a (declarative) Prolog predicate. We found the *black box* constraints in ECL^{i}PS^e and B-Prolog straightforward to use since they were built on Prolog and provided useful high level tools needed for the problem. Oz had an unfair advantage in that we already had an Oz implementation of the SRQ problem. However, compared to clp(FD), SICStus and IF/Prolog, we found that more needed to be learnt before the language could be used effectively. Ilog SOLVER requires a working knowledge of C++ but, since it does not impose any syntactic extension of C++, no new language syntax needs to be learnt. For CHR, not only did we have to learn a new language syntax but also we had to understand the novel constraint propagation mechanism defined by this syntax so that this system took longest for us to master.

3.3 The Self Referential Quiz (SRQ) and Two Solutions

It is well known that the choice of representation can have a dramatic effect on the efficiency of the solution of a problem. Therefore in this section, we describe two alternative formulations for the SRQ puzzle defined in Figure 3.1. The first uses the Boolean domain and represents each option for each question directly as a Boolean variable, requiring fifty such variables. The second requires the finite domain $\{1, 2, 3, 4, 5\}$ and uses an approach similar to the usual representation for the n-queens problem. This

representation requires just ten variables to represent the solution. These two formulations are useful, not only to demonstrate the differences in performance between the Boolean and FD solutions for the different systems but also illustrate two aspects of the meta-reasoning capabilities of the languages: reification and meta-constraints. Note that this is the reason that we do not consider other, possibly more efficient, solutions to the problem[1]. These two aspects are discussed in more detail in the next subsections.

1. The first question whose answer is A is:
   (A) 4 (B) 3 (C) 2 (D) 1 (E) none of the above
2. The only two consecutive questions with identical answers are:
   (A) 3 and 4 (B) 4 and 5 (C) 5 and 6 (D) 6 and 7 (E) 7 and 8
3. The next question with answer A is:
   (A) 4 (B) 5 (C) 6 (D) 7 (E) 8
4. The first even numbered question with answer B is:
   (A) 2 (B) 4 (C) 6 (D) 8 (E) 10
5. The only odd numbered question with answer C is:
   (A) 1 (B) 3 (C) 5 (D) 7 (E) 9
6. A question with answer D:
   (A) comes before this one, but not after this one (B) comes after this one, but not before this one (C) comes before and after this one (D) does not occur at all (E) none of the above
7. The last question whose answer is E is:
   (A) 5 (B) 6 (C) 7 (D) 8 (E) 9
8. The number of questions whose answers are consonants is:
   (A) 7 (B) 6 (C) 5 (D) 4 (E) 3
9. The number of questions whose answers are vowels is:
   (A) 0 (B) 1 (C) 2 (D) 3 (E) 4
10. The answer to this question is:
    (A) A (B) B (C) C (D) D (E) E

Figure 3.1: The SRQ puzzle

3.3.1 Why Self Referential Puzzles and Why These Solutions?

Although there are other kinds of applications of constraint programming over FD, SRQs are particularly appropriate since they incorporate a number of aspects that challenge both the expressiveness and efficiency of the solvers. Our purpose is to evaluate and compare a very wide set of FD and Boolean propagators incorporated in each of the languages studied with the aim of choosing the more adequate to our purposes of generating a generic framework. The two SRQ solutions shown in this chapter require, in particular, a wide and significant set of different propagators over

[1] Such as that suggested by Mark Wallace (personal communication), available in (Fernández, 1998).

$A_1 \equiv A_4 \wedge \neg A_1 \wedge \neg A_2 \wedge \neg A_3$,
$B_1 \equiv A_3 \wedge \neg A_1 \wedge \neg A_2$,
$C_1 \equiv A_2 \wedge \neg A_1$,
$D_1 \equiv A_1$,
$E_1 \equiv \neg A_1 \wedge \neg A_2 \wedge \neg A_3 \wedge \neg A_4$,

$0 \equiv (A_1 \equiv A_2) \wedge (B_1 \equiv B_2) \wedge (C_1 \equiv C_2) \wedge (D_1 \equiv D_2) \wedge (E_1 \equiv E_2)$,
$0 \equiv (A_2 \equiv A_3) \wedge (B_2 \equiv B_3) \wedge (C_2 \equiv C_3) \wedge (D_2 \equiv D_3) \wedge (E_2 \equiv E_3)$,
$A_2 \equiv (A_3 \equiv A_4) \wedge (B_3 \equiv B_4) \wedge (C_3 \equiv C_4) \wedge (D_3 \equiv D_4) \wedge (E_3 \equiv E_4)$,
$B_2 \equiv (A_4 \equiv A_5) \wedge (B_4 \equiv B_5) \wedge (C_4 \equiv C_5) \wedge (D_4 \equiv D_5) \wedge (E_4 \equiv E_5)$,
$C_2 \equiv (A_5 \equiv A_6) \wedge (B_5 \equiv B_6) \wedge (C_5 \equiv C_6) \wedge (D_5 \equiv D_6) \wedge (E_5 \equiv E_6)$,
$D_2 \equiv (A_6 \equiv A_7) \wedge (B_6 \equiv B_7) \wedge (C_6 \equiv C_7) \wedge (D_6 \equiv D_7) \wedge (E_6 \equiv E_7)$,
$E_2 \equiv (A_7 \equiv A_8) \wedge (B_7 \equiv B_8) \wedge (C_7 \equiv C_8) \wedge (D_7 \equiv D_8) \wedge (E_7 \equiv E_8)$,
$0 \equiv (A_8 \equiv A_9) \wedge (B_8 \equiv B_9) \wedge (C_8 \equiv C_9) \wedge (D_8 \equiv D_9) \wedge (E_8 \equiv E_9)$,
$0 \equiv (A_9 \equiv A_{10}) \wedge (B_9 \equiv B_{10}) \wedge (C_9 \equiv C_{10}) \wedge (D_9 \equiv D_{10}) \wedge (E_9 \equiv E_{10})$,

$A_3 \equiv A_4$,
$B_3 \equiv A_5 \wedge \neg A_4$,
$C_3 \equiv A_6 \wedge \neg A_4 \wedge \neg A_5$,
$D_3 \equiv A_7 \wedge \neg A_4 \wedge \neg A_5 \wedge \neg A_6$,
$E_3 \equiv A_8 \wedge \neg A_4 \wedge \neg A_5 \wedge \neg A_6 \wedge \neg A_7$,

$A_4 \equiv B_2$,
$B_4 \equiv B_4 \wedge \neg B_2$,
$C_4 \equiv B_6 \wedge \neg B_2 \wedge \neg B_4$,
$D_4 \equiv B_8 \wedge \neg B_2 \wedge \neg B_4 \wedge \neg B_6$,
$E_4 \equiv B_{10} \wedge \neg B_2 \wedge \neg B_4 \wedge \neg B_6$

$A_5 \equiv C_1 \wedge \neg C_3 \wedge \neg C_5 \wedge \neg C_7 \wedge \neg C_9$,
$B_5 \equiv \neg C_1 \wedge C_3 \wedge \neg C_5 \wedge \neg C_7 \wedge \neg C_9$,
$C_5 \equiv \neg C_1 \wedge \neg C_3 \wedge C_5 \wedge \neg C_7 \wedge \neg C_9$,
$D_5 \equiv \neg C_1 \wedge \neg C_3 \wedge \neg C_5 \wedge C_7 \wedge \neg C_9$,
$E_5 \equiv \neg C_1 \wedge \neg C_3 \wedge \neg C_5 \wedge \neg C_7 \wedge C_9$,

$A_6 \equiv Before^D \wedge \neg After^D$
$B_6 \equiv \neg Before^D \wedge After^D$
$C_6 \equiv Before^D \wedge After^D$
$D_6 \equiv \sum_{i \in \{1 \ldots 10\}} D_i = 0$,
$E_6 \equiv D_6$

$A_7 \equiv E_5 \wedge \neg E_6 \wedge \neg E_7 \wedge \neg E_8 \wedge \neg E_9 \wedge \neg E_{10}$,
$B_7 \equiv E_6 \wedge \neg E_7 \wedge \neg E_8 \wedge \neg E_9 \wedge \neg E_{10}$,
$C_7 \equiv E_7 \wedge \neg E_8 \wedge \neg E_9 \wedge \neg E_{10}$,
$D_7 \equiv E_8 \wedge \neg E_9 \wedge \neg E_{10}$,
$E_7 \equiv E_9 \wedge \neg E_{10}$,

$A_8 \equiv \sum_{i \in \{1 \ldots 10\}} B_i + C_i + D_i = 7$,
$B_8 \equiv \sum_{i \in \{1 \ldots 10\}} B_i + C_i + D_i = 6$,
$C_8 \equiv \sum_{i \in \{1 \ldots 10\}} B_i + C_i + D_i = 5$,
$D_8 \equiv \sum_{i \in \{1 \ldots 10\}} B_i + C_i + D_i = 4$,
$E_8 \equiv \sum_{i \in \{1 \ldots 10\}} B_i + C_i + D_i = 3$,

$A_9 \equiv \sum_{i \in \{1 \ldots 10\}} A_i + E_i = 0$,
$B_9 \equiv \sum_{i \in \{1 \ldots 10\}} A_i + E_i = 1$,
$C_9 \equiv \sum_{i \in \{1 \ldots 10\}} A_i + E_i = 2$,
$D_9 \equiv \sum_{i \in \{1 \ldots 10\}} A_i + E_i = 3$,
$E_9 \equiv \sum_{i \in \{1 \ldots 10\}} A_i + E_i = 4$,

Figure 3.2: SRQ as a satisfiability problem using 50 variables

FD and Boolean variables, important for our comparative study.

All the SRQ solutions compared here are available over Internet (Fernández, 1998).

3.3.2 The Original Idea

The SRQ shown in Figure 3.2 shows the formulation of the SRQ as satisfiability problem. Each question has five options, and each of these options is expressed as a logical formula using the connectives: conjunction, disjunction, negation and equivalence. There are 50 boolean variables $l_i \quad (i \in \{1 \ldots 10\}$ and $l \in \{A, B, C, D, E\})$ where l_i has the value true if the answer to question i is l and false otherwise. Thus we call this formulation the *50 variables formulation*. Note than in Figure 3.2 only 9 of the questions have a formulation. Question 10 is redundant and does not contribute to the solution. The additional variable $Before^D$ is defined to have the truth value of $D_1 \vee D_2 \vee D_3 \vee D_4 \vee D_5$, and $After^D$ is defined to have the truth value of $D_7 \vee D_8 \vee D_9 \vee D_{10}$.

Only one alternative may be true for each question. Thus, we also have the constraints

$$A_j + B_j + C_j + D_j + E_j = 1 \quad (j \in \{1 \ldots 10\}) \tag{3.1}$$

$(Q_1 = 1) \equiv (Q_4 = 1) \wedge \bigwedge_{i \in \{1,2,3\}} (Q_i \neq 1)$,	$(Q_6 = 1) \equiv Before^{Q4} \wedge \neg After^{Q4}$,
$(Q_1 = 2) \equiv (Q_3 = 1) \wedge \bigwedge_{i \in \{1,2\}} (Q_i \neq 1)$,	$(Q_6 = 2) \equiv \neg Before^{Q4} \wedge After^{Q4}$,
$(Q_1 = 3) \equiv (Q_2 = 1) \wedge (Q_1 \neq 1)$,	$(Q_6 = 3) \equiv Before^{Q4} \wedge After^{Q4}$,
$(Q_1 = 4) \equiv (Q_1 = 1)$, $^{(*)}$	$(Q_6 = 4) \equiv \bigwedge_{i \in \{1...10\}} (Q_i \neq 4)$,
$(Q_1 = 5) \equiv \bigwedge_{i \in \{1,2,3,4\}} (Q_i \neq 1)$,	$(Q_6 = 5) \equiv (Q_6 = 4)$, $^{(*)}$
$(Q_2 = 1) \equiv (Q_3 = Q_4)$,	$(Q_7 = 1) \equiv (Q_5 = 5) \wedge \bigwedge_{i \in \{6...10\}} (Q_i \neq 5)$,
$(Q_2 = 2) \equiv (Q_4 = Q_5)$,	$(Q_7 = 2) \equiv (Q_6 = 5) \wedge \bigwedge_{i \in \{7...10\}} (Q_i \neq 5)$,
$(Q_2 = 3) \equiv (Q_5 = Q_6)$,	$(Q_7 = 3) \equiv (Q_7 = 5) \wedge \bigwedge_{i \in \{8...10\}} (Q_i \neq 5)$, $^{(*)}$
$(Q_2 = 4) \equiv (Q_6 = Q_7)$,	$(Q_7 = 4) \equiv (Q_8 = 5) \wedge \bigwedge_{i \in \{9...10\}} (Q_i \neq 5)$,
$(Q_2 = 5) \equiv (Q_7 = Q_8)$,	$(Q_7 = 5) \equiv (Q_9 = 5) \wedge (Q_{10} \neq 5)$,
$(Q_3 = 1) \equiv (Q_4 = 1)$,	$(Q_8 = 1) \equiv \sum_{i \in \{1...10\}} BCD_i = 7$,
$(Q_3 = 2) \equiv (Q_5 = 1) \wedge (Q_4 \neq 1)$,	$(Q_8 = 2) \equiv \sum_{i \in \{1...10\}} BCD_i = 6$,
$(Q_3 = 3) \equiv (Q_6 = 1) \wedge \bigwedge_{i \in \{4,5\}} (Q_i \neq 1)$,	$(Q_8 = 3) \equiv \sum_{i \in \{1...10\}} BCD_i = 5$,
$(Q_3 = 4) \equiv (Q_7 = 1) \wedge \bigwedge_{i \in \{4,5,6\}} (Q_i \neq 1)$,	$(Q_8 = 4) \equiv \sum_{i \in \{1...10\}} BCD_i = 4$,
$(Q_3 = 5) \equiv (Q_8 = 1) \wedge \bigwedge_{i \in \{4,5,6,7\}} (Q_i \neq 1)$,	$(Q_8 = 5) \equiv \sum_{i \in \{1...10\}} BCD_i = 3$,
$(Q_4 = 1) \equiv (Q_2 = 2)$,	$(Q_9 = 1) \equiv \sum_{i \in \{1...10\}} AE_i = 0$,
$(Q_4 = 2) \equiv (Q_4 = 2) \wedge (Q_2 \neq 2)$,	$(Q_9 = 2) \equiv \sum_{i \in \{1...10\}} AE_i = 1$,
$(Q_4 = 3) \equiv (Q_6 = 2) \wedge \bigwedge_{i \in \{2,4\}} (Q_i \neq 2)$,	$(Q_9 = 3) \equiv \sum_{i \in \{1...10\}} AE_i = 2$,
$(Q_4 = 4) \equiv (Q_8 = 2) \wedge \bigwedge_{i \in \{2,4,6\}} (Q_i \neq 2)$,	$(Q_9 = 4) \equiv \sum_{i \in \{1...10\}} AE_i = 3$,
$(Q_4 = 5) \equiv (Q_{10} = 2) \wedge \bigwedge_{i \in \{2,4,6,8\}} (Q_i \neq 2)$,	$(Q_9 = 5) \equiv \sum_{i \in \{1...10\}} AE_i = 4$,
$(Q_5 = 1) \equiv (Q_1 = 3) \wedge \bigwedge_{i \in \{3,5,7,9\}} (Q_i \neq 3)$,	To enforce the 'only' part of question 2
$(Q_5 = 2) \equiv (Q_3 = 3) \wedge \bigwedge_{i \in \{1,5,7,9\}} (Q_i \neq 3)$,	$Q_1 \neq Q_2,\ Q_2 \neq Q_3$
$(Q_5 = 3) \equiv (Q_5 = 3) \wedge \bigwedge_{i \in \{1,3,7,9\}} (Q_i \neq 3)$,	$Q_8 \neq Q_9,\ Q_9 \neq Q_{10}$
$(Q_5 = 4) \equiv (Q_7 = 3) \wedge \bigwedge_{i \in \{1,3,5,9\}} (Q_i \neq 3)$,	
$(Q_5 = 5) \equiv (Q_9 = 3) \wedge \bigwedge_{i \in \{1,3,5,7\}} (Q_i \neq 3)$,	

Figure 3.3: SRQ as a satisfiability problem using 10 variables

The problem consists in finding an assignment of the variables A_i, B_i, C_i, D_i and E_i ($i \in \{1 \ldots 10\}$) to truth values such that formula 3.1 and all the formulas in Figure 3.2 hold. This formulation was translated to an Oz program in (Henz, 1996). In Figure 3.4 the table to the left shows the solution to SRQ by this approach.

3.3.3 An Alternative Approach

A more compact representation would have a single variable for each question and assign the code for the correct answer for that question to the variable (in the style of the usual representation for the n-queens problem). For that reason, a formulation for SRQ involving only 10 FD variables, which we call the *10 variables formulation* and shown in Figure 3.3, is studied here. There is exactly one FD variable Q_i ($i \in \{1, \ldots, 10\}$) for each of the ten questions. Each Q_i takes a value in the domain 1..5 such that the answer to the *i'th* question is in the position Q_i in the list $[A, B, C, D, E]$. The problem comprises finding a value for each of the Q_i such that all the formulas in Figure 3.3 hold. The additional variable BCD_i (in the formula for question 8) is 1 if $Q_i \in \{2, 3, 4\}$ and 0 otherwise and AE_i (in the formula for question 9) is 1 if $Q_i \in \{1, 5\}$ and 0 otherwise

($i \in \{1 \ldots 10\}$). The variable $Before^{Q4}$ has the truth value of $\bigvee_{i \in \{1 \ldots 5\}}(Q_i = 4)$, and $After^{Q4}$ the truth value of $\bigvee_{i \in \{7 \ldots 10\}}(Q_i = 4)$. The table to the right in Figure 3.4 shows the solution by this approach.

	A	B	C	D	E	Q
1	0	0	1	0	0	3
2	1	0	0	0	0	1
3	0	1	0	0	0	2
4	0	1	0	0	0	2
5	1	0	0	0	0	1
6	0	1	0	0	0	2
7	0	0	0	0	1	5
8	0	1	0	0	0	2
9	0	0	0	0	1	5
10	0	0	0	1	0	4

Figure 3.4: Solutions to SRQ using 50 and 10 variables

Note that the constraints marked with (*) in Figure 3.3 are obviously inconsistent because they constrain an FD variable to have more than one value at the same time. We observe that such constraints can lead to the removal of inconsistent values from the domain of the variables before any choices are made (meaning a priori-pruning in the search space).

The 10 variables formulation uses a form of meta-constraint, that is, a constraint over constraints. This means that the constraint logical connectives are applied on constraint expressions where a constraint expression is either an arithmetic constraint (in the way of $Q_i = n$) or a combination of constraint expressions using the logical FD connectives.

3.4 Reification and Meta-constraints

In Section 2.7 we introduce a class of complex constraints that lead to provide higher flexibility to the codification of CSPs. In this class we group the reified constraints and meta-constraints. In this section we explain how these can be coded in each of the eight languages involved in the comparison: clp(FD), ECL^{i}PS^e, Oz, SICStus, IF/Prolog, Ilog SOLVER, B-Prolog and CHR. The first seven languages are analysed in Section 3.4.1. The CHR language is considered separately in Section 3.4.2 since the CHR language is so flexible that CHR code can be written in each of the styles of the other languages. We use the SRQ puzzle to illustrate this.

It is important to note that we only consider one aspect of the expressiveness (that concerned with reification and meta-constraints). More comprehensive discussions concerning expressiveness should also consider, among others, aspects such as (a) the ability to express search strategies, (b) the ease of development and integration of constraint satisfaction techniques and (c) the ease of reusing previous work through

the development and exploitation of additional libraries.

3.4.1 Expressing Reification and Meta-constraints

We discuss here the means by which each language can be used to express meta-constraints and reified constraints.

EXAMPLE 3.1 *The logical formula associated to option A in question 6 in the* 50 variables formulation *for SRQ is expressed, using reified constraints, as follows:*

$$A_6 \equiv BfD \wedge \neg AfD \tag{3.2}$$

where the variable BfD is defined to have the truth value of $D_1 \vee D_2 \vee D_3 \vee D_4 \vee D_5$, *and AfD is defined to have the truth value of* $D_7 \vee D_8 \vee D_9 \vee D_{10}$.

The logical formula associated to option A in question 6 in the 10 variables formulation *for SRQ is expressed, using meta-constraints, as follows:*

$$(Q_6 = 1) \equiv BfQ \wedge \neg AfQ, \tag{3.3}$$

where the variable BfQ has the truth value of $\bigvee_{i\in\{1...5\}}(Q_i = 4)$, *and AfQ the truth value of* $\bigvee_{i\in\{7...10\}}(Q_i = 4)$.

Formulas 3.2 and 3.3 are used to highlight the differences between the languages.

Oz, SICStus and IF/Prolog.

• These languages allow a reified constraint form and admit propagators with concatenation in the way of $arg_0\ R_1\ arg_1\ R_2 \ldots R_{n-1}\ arg_{n-1}\ R_n\ arg_n$, where each R_i is a propagator and arg_{i-1} and arg_i are arguments ($i \in \{1 \ldots n\}$). For instance, the disjunction propagator $\vee$ can be concatenated in the way of $D_1 \vee D_2 \vee D_3 \vee D_4 \vee D_5$ where each D_i ($1 \leq i \leq 5$) is a Boolean variable.

• Propagators can operate directly over Boolean variables (i.e. $B_1 \vee B_2$ or $\sim B_1$ where B_1 and B_2 are Boolean variables and $\vee$ and $\sim$ represent the disjunction and negation propagators respectively[2]) or over constraint expressions (i.e. $(Q_1 = n_1) \vee (Q_2 = n_2)$ where Q_1 and Q_2 are FD variables and the values n_1 and n_2 belong to their domains).

• Meta-constraints can be combined with reified constraints as for instance in $B \equiv (Q_1 = n_1) \wedge (Q_2 = n_2)$ where B is a Boolean variable.

ECL^{i}PS^e and Ilog SOLVER.

• The propagators can be concatenated as in Oz, SICStus or IF/Prolog. The main difference with these languages is that some propagators are limited to operate on constraint expressions over FD variables and not directly on the variables themselves. The direct application of the constraint propagators such as disjunction, conjunction

[2]The syntax of the propagator changes for each language.

and negation over the Boolean variables is not allowed. For instance, to express that the negation of a Boolean variable B must be true we must write $(B\# = 0)$ or $(B\backslash\# = 1)$ in ECL^{i}PS^e (resp. $(B == 0)$ or $!(B == 1)$ in Ilog).

- ECL^{i}PS^e does not allow the use of reified constraints.
- Ilog SOLVER allows the reification for the Boolean variables.

clp(FD) and B-Prolog.

- Most of the FD constraint propagators have a relational form (that is, in the way of *R(X,Y,Result)*, where R is a propagator and X, Y and *Result* are FD Boolean variables) close to the style of traditional Prolog. Thus it is not possible to employ the concatenation shown in other languages (which means that, in many cases, large numbers of extra variables are needed).
- The Boolean operators also require a relational form which means that the code needs a large number of extra FD variables.
- Meta-constraints cannot be implemented in clp(FD) in a direct way. For instance, the constraint imposed for the variable *BfQ* in formula 3.3 which must have the truth value of $\bigvee_{i\in\{1\ldots5\}}(Q_i = 4)$ is best expressed as follows:

$$(BfQ = 1) \equiv (Q1 = 4) \vee (Q2 = 4) \vee (Q3 = 4) \vee (Q4 = 4) \vee (Q5 = 4). \tag{3.4}$$

Code is now required that detects when the constraint $(BfQ = 1)$ is true. It is explained in (Carlson et al., 1994a) how the clp(FD) framework can be extended so that constraints such as this can be expressed using just constraints of the form *X in r*. However, as this idea is not implemented yet (at least in the version of the language used here), the approach in (Codognet and Diaz, 1996a) for solving the magic square problem can be adopted. For instance, for the constraint (3.4) the predicate '$x = a\Leftrightarrow b$'/3 can be used. The call '$x = a\Leftrightarrow b$'(X, A, B) means $X = A$ *iff* B is true (i.e. $B = 1$) and is defined:

$$`x = a\Leftrightarrow b`(X, A, B) :- B \text{ in } `X\_To\_B`(dom(X), A),\ X \text{ in } `B\_To\_X`(val(B), A).$$

where 'X_To_B' returns 1 if $X = A$, 0 if $X \neq A$ and 0..1 otherwise; and 'B_To_X', which is delayed until B is instantiated, yields A if $B = 1$ or else the range $0..\infty\backslash A$. These *user functions* are written in C and can accept ranges or terms as argument. In clp(FD) the constraint (3.4) can now be expressed as follows:

$$`x = a\Leftrightarrow b`(BfQ, 1, B1),\ `x = a\Leftrightarrow b`(Q1, 4, B2),\ `x = a\Leftrightarrow b`(Q2, 4, B3),$$
$$`x = a\Leftrightarrow b`(Q3, 4, B4),\ `x = a\Leftrightarrow b`(Q4, 4, B5),\ `x = a\Leftrightarrow b`(Q5, 4, B6),$$
$$or(B2, B3, B23),\ or(B4, B5, B45),\ or(B23, B45, B2345),\ or(B2345, B6, B1).$$

Note that, just for this one constraint, nine additional Boolean variables (B1, B2, B3, B4, B5, B6, B23, B45, B2345) are necessary.

- In B-Prolog, as for clp(FD), meta-constraints cannot be defined directly, but the reification of Boolean variables can be expressed by means of delay clauses (which

avoids the need for any C code). This can be done by defining a predicate *iff*/3[3] as follows:

$$\begin{aligned}
&delay\ iff(X,Y,B)\ :-\ dvar(B), dvar(X)\ :\ true.\\
&delay\ iff(X,Y,B)\ :-\ dvar(B), dvar(Y)\ :\ true.\\
&\quad iff(X,Y,B)\ :-\ integer(B)\ :\ (B =:= 1 -> \ X\#=Y;\ X\#\backslash=Y).\\
&\quad iff(X,Y,B)\ :-\ X =:= Y\ :\ B = 1.\\
&\quad iff(X,Y,B)\ :-\ true\ :\ B = 0.
\end{aligned} \tag{3.5}$$

Note that #= and #\ are the equality and disequality propagators respectively. The part of the body of the clauses before the : is called a guard. The first two clauses are called *delay* clauses which have guards of the form $dvar(Z)$ which checks that its argument Z is a domain variable. As a result of the $delay/1$ clauses, the *iff*/3 clauses are only executed when either B or X and Y are instantiated to unique values. The remaining three clauses define *iff*/3. These use guards $integer(B)$, $X =:= Y$ and *true*. Once the guard of a clause has succeeded, the remaining clauses defining *iff*/3 are discarded and the call becomes determinate. The call *iff(X,Y,B)* means $(X = Y) \Leftrightarrow B$, that is, if B is true, then the constraint $X\#= Y$ is imposed and, if B is false, the disequality $X\backslash\# = Y$ is imposed. On the other hand, if the constraint $X = Y$ is true (false), then B is imposed to be true (false). The constraint (3.4) is then coded as follows:

$$\begin{aligned}
&iff(BfQ,1,B1), iff(Q1,4,B2), iff(Q2,4,B3), iff(Q3,4,B4),\\
&\qquad iff(Q4,4,B5), iff(Q5,4,B6), or(B2,B3,B23),\\
&or(B4,B5,B45), or(B23,B45,B2345), or(B2345,B6,B1).
\end{aligned}$$

3.4.2 CHR: a Special Mention

Expressively speaking, CHR deserves a special consideration due to its flexibility for writing solvers. Its particular *glass box* approach allows one to provide the same formulation in different styles. To demonstrate this flexibility, the clp(FD) language can be used as a model for writing the formula 3.2 in CHR code. The different clp(FD) Boolean propagators can easily be simulated by the CHR constraints.

Meta-constraints and *reified* constraints are no problem in CHR. For instance, the formula 3.3 can be coded in CHR in a style close to Oz. This can be done by defining an equivalence propagator between constraint expressions by means of the definition of a *solve/2* predicate which receives in its first argument a logical restriction over variable constraints (that is a meta-constraint) or FD variables and returns in its second argument the result of it. To impose a constraint it is enough to have the value 1 (denoting true) in the last argument and the constraint in the first argument. Then,

[3]Personal communication with Neng-Fa Zhou.

the formula 3.3 can be expressed in CHR as follows:

$$solve(BfQ \Leftrightarrow (Q1 = 4) + (Q2 = 4) + (Q3 = 4) + (Q4 = 4) + (Q5 = 4), 1),$$
$$solve(AfQ \Leftrightarrow (Q7 = 4) + (Q8 = 4) + (Q9 = 4) + (Q10 = 4), 1),$$
$$solve((Q6 = 1) \Leftrightarrow (BfQ* \sim AfQ), 1).$$

The first argument of the *solve* predicate simulates, with minor differences, the Oz code employed to program the formula 3.3. Note that the Oz arithmetic propagators $*$, $+$ and $\sim$ have been defined as the Boolean propagators conjunction, disjunction and negation respectively (as it was done in the Oz program for the SRQ).

3.5 An Efficiency Comparison

In this section we compare the efficiency of the eight constraint systems described in Section 3.2. Subsection 3.5.1 describes the two labeling strategies used and 3.5.2 gives the results for the solving of the SRQ in Figure 3.1 under the two different approaches described in Section 3.3. The efficiency study is extended to other benchmarks in Section 3.5.3 and the section concludes with an evaluation of the results.

Note that all of the systems (except CHR) provide some built-in symbolic constraints which we used in the benchmarks' code. In particular, for all systems for which it was provided, the *all_different* constraint was employed where applicable. Thus although we tried to maintain the same formulation for any given benchmark across all the systems, we did exploit all the facilities that were provided by a system to obtain the best possible results.

3.5.1 Labeling

As it was shown in Section 2.2.3, constraint solving can be seen as a combination of two processes, constraint propagation and labeling. Constraint propagation is a procedure that reads the constraint store and imposes constraints to it by means of constraint propagators (Jaffar and Lassez, 1987). Labeling assigns values to the domain variables (instantiation). Thus the labeling process consists of (1) choosing a variable (variable ordering) and (2) assigning to the variable a value belong to its domain (value ordering). The variable ordering and the value ordering used for the labeling can considerably influence the efficiency of the constraint solving when only one solution to the problem is required. It has little effect when the search is for all solutions. In this study, we considered two labelings that were described in Example 2.3 on page 17, the *naive* labeling and the *first fail* labeling.

All the results in Section 3.5.2 were obtained using the *first fail* labeling. As far as possible, we have maintained the same variable and value ordering for all the systems in the efficiency comparison. We also used a *naive labeling*. In contrast to *first fail*, *naive* labeling assures that both variable and value ordering are the same for all the systems and hence in many ways, although less efficient, is better for comparing the different systems when only one solution is required. Thus additional results for the

first solution search were obtained using the *naive* labeling. The performance results for this are given in Section 3.5.3.

3.5.2 Efficiency Compared on the SRQ

In this subsection, we discuss the efficiency of the eight systems for solving the SRQ. We consider both of the formulations described in Section 3.3. Thus first we examine how the choice of representation affects the performance and hence the efficiency results. The performance of the systems for the SRQ are then compared, first under the 50 variables formulation and then under the 10 variables formulation.

Comparing the Search Trees

Figure 3.5: The Oz Explorer on (50 variable formulation) SRQ solving using *first fail* labeling for *first solution* search

The hardness of this kind of puzzle seems to be in the number of choice-points that are traversed in finding a solution. For that reason, we compare the different Oz programs under the two different approaches (50 and 10 variables) giving them

Figure 3.6: The Oz Explorer on (10 variable formulation) SRQ solving using *first fail* labeling for *first solution* search

to a particular inference machine (which performs the search) called Explorer tool (Schulte, 1995). This allows the search tree to be visualised as shown in Figures 3.5 and 3.6. Here the choice nodes are denoted by circles, the failure nodes by squares and the solution nodes by diamonds. The tree in Figure 3.5 shows the search tree of the original Oz program for *first solution* search in the SRQ solving using *first fail* labeling (see Subsection 3.5.1). This tree contains 27 nodes of which 13 are choice nodes. The tree in Figure 3.6 shows, under the same conditions, the search tree for the 10 variables formulation which contains 9 nodes with only 5 choice nodes. This is more than a 60% reduction in the number of choice nodes.

The SRQ Results Using the Boolean Domain

Here, we present the performance of the set of programs for the SRQ implemented under the approaches shown in Section 3.3.2. The original solution in Oz involves 50 Boolean variables and all the other programs have been implemented similarly. We modified the labeling strategy for the original Oz program[4] because we wanted to do the efficiency comparisons under the same conditions of labeling. The *first fail* labeling

[4]When no more propagation was possible, the variable on which more propagators depend was chosen, and then its maximal value was tried first. In this case, the most suitable strategy was used

(see Example 2.3 on page 17) was used for all the programs except for the CHR one which used the built-in labeling for the constraint handling rules.

Since we were not able to install all the systems on the same machine, we used two machines, a 4/50 SPARCstation IPX (40 MHz) and a Pentium Pro 200 PC operating under Linux. The SRQ programs for ECL^{i}PS^e, CHR (available as a library of ECL^{i}PS^e), clp(FD), Ilog SOLVER, B-Prolog, and Oz, were measured using the SPARCstation and programs for SICStus, IF/Prolog, and again ECL^{i}PS^e, have all been run on the PC. Note that the program for ECL^{i}PS^e was measured on both machines to provide a means of comparing the results across all platforms.

Table 3.1 summarises the results for the programs that used the SPARCstation while Table 3.2 gives the results for the programs using the PC.

Table 3.1: Performance results of the 50 variables formulations for the SRQ on Sparc 40 Mhz

language	*time first*	*time all*	*speedup*
Ilog SOLVER	80	100	3.63
clp(FD)	80	110	3.63
ECL^{i}PS^e	933	1083	↓ 3.22
CHR	6150	-	↓ 21.21
B-Prolog	217	270	1.34
Oz	290	305	1.00

Table 3.2: Performance results of the 50 variables formulations for the SRQ on PC (Linux)

language	*time first*	*time all*	*speedup*
SICStus	40	50	3.00
IF/Prolog	20	20	6.00
ECL^{i}PS^e	120	150	1.00

The meaning for the columns is as follows. The first column gives the name of the constraint language used in the implementation. The second column gives the running time to find the first (and unique) answer, measured in milliseconds. The next column shows the time to explore the whole search space. The last column gives the average speed-up. In Table 3.1, this is in relation to the original program in Oz[5] whereas in Table 3.2 it is in relation to ECL^{i}PS^e. The symbol ↓ in Table 3.1 indicates that the following number is the average slow-down instead of speed-up with respect to Oz. Because the slow-down for CHR is so high we have only provided the result for the *first solution* search.

To give an idea of the efficiency of each of the systems with respect to each other, Table 3.3 shows the speedup of each system in relation to ECL^{i}PS^e (which is taken as

and the general condition could propagate much better.

[5]Note that Oz has been taken as reference since it provided the original solution.

reference).

Table 3.3: Result normalisation for 50 variables formulations on the ECL^{i}PS^e column

Ilog	*clp(FD)*	*Oz*	*CHR*	*SICStus*	*IF/Prolog*	*ECL^{i}PS^e*	*B-Prolog*
11.7	11.7	3.2	↓ 6.6	3.0	6.0	1.0	4.3

Table 3.4: Comparable results of the 10 variables formulation for the SRQ on Sparc 40 Mhz

language	*time first*	*time all*	*speedup*	*speedup prec*
Ilog SOLVER	40	90	7.25	2.00
clp(FD)	30	40	9.67	2.67
ECL^{i}PS^e	583	933	↓ 2.01	1.60
CHR	4400	-	↓ 15.17	1.39
B-Prolog	33	50	8.79	6.58
Oz	186	295	1.56	1.56

Table 3.5: Comparable results of the 10 variables formulation for the SRQ on PC(Linux)

language	*time first*	*time all*	*speedup*	*speedup prec*
SICStus	30	40	4.00	1.33
IF/Prolog	10	10	12.00	2.00
ECL^{i}PS^e	100	140	1.20	1.20

These results show that for the SRQ problem, the Ilog and clp(FD) programs are fastest, while Oz, SICStus, IF/Prolog and B-Prolog were about two to four times slower. However, these latter four systems were at least three times as fast as ECL^{i}PS^e. Of course, as the times needed to solve the SRQ problem are not high, the real differences in the performance of these systems are not necessarily indicated here (in the next subsection, we compare again the performance of these systems using a larger set of benchmarks). The CHR program is by far the slowest. This is to be expected since the CHR used here is implemented on the top of the ECL^{i}PS^e system.

The SRQ Results Using the 10 Variables Approach

Using the 10 variables formulation in Section 3.3.3, we implemented a program for all the languages. As for the 50 variables approach the results were measured on two different machines and later normalised by the ECL^{i}PS^e results. Tables 3.4 and 3.5 show the results and Table 3.6 shows the speedup of each of the systems with respect to the ECL^{i}PS^e system.

The meanings for the first four columns in Tables 3.4 and 3.5 are the same as in Tables 3.1 and 3.2. Just as in the 50 variables formulation, only the result for the

Table 3.6: Result normalisation for 10 variables formulation on the column ECL^{i}PS^e

Ilog	*clp(FD)*	*Oz*	*CHR*	*SICStus*	*IF/Prolog*	*ECL^{i}PS^e*	*B-Prolog*
14.6	19.4	3.1	↓ 7.5	3.3	10.0	1.0	17.7

first solution search is given for CHR. Note that, in Table 3.4, the column *speedup* gives the speed-up with respect to the original program in Oz (that is, with respect to the 50 variables formulation in Oz) and, in Table 3.5, with respect to the ECL^{i}PS^e program with 50 variables. The fifth column *speedup prec* indicates the speed-up factor with respect to the 50 variables program implemented in the same language (results in Tables 3.1 and 3.2). All programs implemented under the 10 variables approach improve the speed of the programs over those implemented in the style of the original Oz program involving 50 Boolean variables. This illustrates how a change of representation may significantly affect the performance.

Table 3.6 compares all the programs for the 10 variables formulation by normalising the results relative to the ECL^{i}PS^e timings.

3.5.3 A More Comprehensive Comparison

Some Extra Benchmarks

It is clear that a single benchmark may bias the performance results unfairly. Typically, few constraint languages dominate such an application and thus the resulting performance figures are vulnerable to slight variations of the implementation of these constraints. In order to make the comparison more objective we extended the work to include the following well-known benchmarks (Van Hentenryck, 1989):

- **sendmore**: a cryptarithmethic problem on 8 variables ranging over 0..9, with one linear equation and 36 disequations;
- **alpha**: a cipher problem involving 26 variables over 1..26, with 20 equations and 325 disequations;
- **equation 10**: a system of 10 linear equations with 7 variables over 0..10;
- **equation 20**: a system of 20 linear equations with 7 variables over 0..10;
- **N queens**: place N queens on a $N \times N$ chessboard in such a way that no queen attacks each other;
- **magic sequences (N)**: calculate a sequence of N numbers such that each of them is the number of occurrences in the series of its position in the sequence.

The programs **sendmore**, **alpha**, **equation 10** and **equation 20** test the efficiency of the systems to solve linear equation problems. The **N queens** and **magic sequences** programs are scalable and therefore useful to test how the systems works for bigger instances of the same problem. Note that both the number of variables and

Table 3.7: Performance results on Sparc (25 MHz) for first solution search.

Benchmark	*Ilog*	*clp(FD)*	*Oz*	*CHR*	*ECL^{i}PS^e*	*B-Prolog*
SRQ (10)	0.060	0.050	0.290	7.600	0.900	0.050
SRQ (50)	0.120	0.120	0.410	9.300	1.380	0.335
64 queens	0.110	2.710	4.190	-	10.816	0.200
100 queens	0.230	*Error1*	1.270	-	5.533	119.300
sendmore	0.010	0.020	0.020	-	0.033	0.010
alpha	0.130	0.300	0.480	-	1.633	0.233
eq. 10	0.200	0.270	0.460	-	0.833	0.367
eq. 20	0.240	0.360	0.460	-	1.050	0.817
magic(10)	0.020	0.040	0.100	-	0.333	0.284
magic(50)	0.110	1.090	1.870	-	8.234	5.300
magic(100)	0.280	4.320	16.290	-	49.700	*Error4*
magic(130)	0.410	*Error1*	35.030	-	*Error3*	*Error4*
magic(150)	0.530	*Error1*	50.790	-	*Error3*	*Error4*
magic(200)	0.860	*Error1*	*Error2*		*Error3*	*Error4*

the number of values for each variable grow linearity with N. That is, given a value N, at least N FD variables must be declared with domains that range between 0 or 1 and N.

The programs used for the measurements for each of the extra benchmarks listed above were either provided with the system or first written by us but then improved by the language designers themselves. This policy meant that, for each system, only appropriate programs have been compared. We measured the time required both for finding just one solution and, where possible, in finding all solutions.

As observed in Section 3.5.1, the choice of labeling can affect the performance when searching for just one solution. Thus, for just the one solution case, in this section the systems have been compared using both the *first fail* labeling and the *naive* labeling.

For ECL^{i}PS^e, we initially used the **N queens** program provided with the system and an adaptation of the SICStus program for the **magic sequences**. However, the performance was poor. We found that one of the reason for the inefficiency was the *first fail* labeling. When there are several with the same smallest domain, the one chosen is not defined and dependent on the implementation. In order to improve the efficiency, the *first fail* labeling of Joachim Schimpf[6] was used which was found to improve the speed by a factor of 14. To obtain further improvements, the programs were compiled without debugging information and without garbage collection[7]. As a result of this, for the **magic sequences** program, we improved the speed by a factor of 70. However, the lack of garbage collection reduced the size of problems that could be solved. It should also be noted that removing the garbage collection in other systems such as SICStus did not improve the performance.

[6]Personal communication with Joachim Schimpf.

[7]Personal communication with Mark Wallace and Joachim Schipmf.

Table 3.8: Performance results on Sparc (25 MHz) for all solutions search.

Benchmark	*Ilog*	*clp(FD)*	*Oz*	*ECL^{i}PS^e*	*B-Prolog*
8 queens	0.340	0.370	2.210	2.167	0.200
9 queens	1.350	1.410	10.200	8.950	0.817
10 queens	5.350	5.270	36.540	34.467	3.216
11 queens	23.920	23.970	178.240	166.583	14.700
12 queens	118.200	119.760	836.891	840.650	72.533
magic(30)	0.160	0.890	0.890	5.050	6.300
magic(75)	1.300	8.680	9.860	55.850	102.734
magic(100)	2.690	18.380	24.580	125.600	*Error4*
magic(130)	5.300	*Error1*	48.590	*Error3*	*Error4*
magic(150)	7.640	*Error1*	83.780	*Error3*	*Error4*

The benchmarks measured with *first fail* labeling

All the results presented in Tables 3.7 to 3.12 were obtained using the *first fail* labeling. For each of the benchmarks, the times needed to obtain the first solution have been measured with each of the constraint systems except for CHR. For CHR, the only benchmarks used were the 50 and 10 variables solutions for the SRQ. This is because the only solutions that could be found for the other benchmarks were extremely inefficient; no solutions for these were supplied with the CHR library and running the existing ECL^{i}PS^e code with the CHR finite domain library (as suggested by the CHR author[8]) was still too slow. As an excuse for this, we observe that CHR was not built for writing efficient solvers but for defining adequate constraints solvers for particular problems on specific domains.

For the scalable problems (that is, the **N queens** and **magic sequences**), the times for finding all possible solutions have also been measured with each of the constraint systems (except, that is, for CHR).

As explained in Section 3.5.2, since we were not able to install all the systems on the same machine, measurements for clp(FD), CHR, Ilog, Oz and B-Prolog have been obtained on a different machine to SICStus and IF/Prolog. Programs for ECL^{i}PS^e were timed on both machines so that the results could be compared. All the timings are in seconds.

Tables 3.7 and 3.8 show the results with Ilog SOLVER, clp(FD), Oz, CHR, ECL^{i}PS^e, and B-Prolog, using the same SPARCstation IPC 4/40 to 25 Mhz[9]. Table 3.7 gives the times for finding the first solution and Table 3.8 the times for obtaining all solutions for the **N queens** and **magic sequences** problems.

Tables 3.9 and 3.10 show the results with SICStus, IF/Prolog, and again ECL^{i}PS^e, using a Pentium Pro 200 PC operating under LINUX. Table 3.9 gives the times for

[8]Personal communication with Thom Frühwirth.

[9]Note this machine is different to that used to measure results on SRQ solving in Section 3.5.2.

Table 3.9: Performance results on PC (Linux) for first solution search.

Benchmark	*SICStus*	*IF/Prolog*	*ECL^{i}PS^e*
SRQ (10)	0.030	0.010	0.100
SRQ (50)	0.040	0.020	0.120
64 queens	0.500	0.300	0.820
100 queens	0.460	0.430	0.460
sendmore	0.005	0.005	0.010
alpha	0.065	8.090	0.120
eq. 10	0.045	0.010	0.055
eq. 20	0.050	0.010	0.070
magic(10)	0.020	0.020	0.030
magic(50)	0.280	0.290	0.650
magic(100)	1.080	1.570	3.830
magic(130)	1.820	2.910	7.750
magic(150)	2.330	3.700	*Error3*
magic(200)	4.180	13.450	*Error3*

finding the first solution and Table 3.10 shows the times for obtaining all the solutions for the **N queens** and **magic sequences** problems.

In these tables, *Error1* in the clp(FD) columns means that the error message "trail stack overflow" was returned. This can be avoided by increasing the size of the environment variable associated with the stack, although the solution is very slow. The *Error2* in the Oz column is because Oz system "died" before solving the problem. The *Error3* in the ECL^{i}PS^e columns means that there was a stack error. *Error4* in the B-Prolog column means a *Trail* or a *Control Stack Overflow* error was received. The anomalous results in Tables 3.7 and 3.9 where, for Oz, SICStus and ECL^{i}PS^e, **64 queens** took longer than the **100 queens** are due to the choice of *first fail* labeling. The results shown in Section 3.5.3 using the *naive* labeling are more consistent.

The results shown in Tables 3.7 to 3.10 were normalised by means of the ECL^{i}PS^e timings. The normalisation of results for first solution search is shown in Table 3.11 and for all solutions search is shown in Table 3.12. Each cell contains the speedup with respect to the ECL^{i}PS^e solution. Again the symbol $\downarrow$ means that the following number is the average slow-down with respect to ECL^{i}PS^e system. In the cases in which ECL^{i}PS^e returned an error, we calculated the average differences between the two machines used to measure the results and normalised the results for IF/Prolog. Note that this only occurs in the last two rows of Tables 3.11 and 3.12.

The benchmarks measured with *naive* labeling

The efficiency results using the *naive* labeling (see Subsection 3.5.1) are shown in Tables 3.13, 3.14 and 3.15. Note that, for the **N queens** problem, no results are shown because the running times were too high. Only results for first solution search are shown. The

Table 3.10: Performance results on PC (Linux) for all solutions search.

Benchmark	*SICStus*	*IF/Prolog*	*ECL^{i}PS^e*
8 queens	0.130	0.040	0.180
9 queens	0.510	0.180	0.740
10 queens	2.000	0.700	2.950
11 queens	8.810	3.150	13.340
12 queens	43.120	15.580	66.150
magic(30)	0.160	0.140	0.420
magic(75)	1.050	1.140	4.140
magic(100)	1.930	2.490	9.100
magic(130)	3.360	4.720	*Error3*
magic(150)	4.600	7.050	*Error3*

results for all solutions search were similar to those shown using the *first fail* labeling.

Robustness

Using the same machines as for the efficiency comparison in Section 3.5.3, the robustness of each the systems was measured. For this, we used the **magic sequences (N)** programs (with garbage collection on) and measured the maximum value of **N** that each system could manage. Tables 3.16 and 3.17 give an interval of FD variables. Each system succeeded at the minimum of the interval while it failed at the maximum. Note that, for both machines, there was no upper bound for ECL^{i}PS^e. This was due to the garbage collection which made these tests for ECL^{i}PS^e extremely slow. CHR was not evaluated. In spite of the different configurations of the two machines used, the results on them for ECL^{i}PS^e are almost the same.

Note that the precise values in these results are very dependent on the machine used. However, these results do provide some indication of the comparative robustness of the systems.

3.5.4 The Results Analysed

In this section we summarise and compare the performance results provided in Sections 3.5.2 and 3.5.3.

Ilog SOLVER. In general, Ilog was by far the fastest system. Ilog was also extremely robust, solving the **magic sequences** problem with over 1600 variables.

clp(FD). This gave good results, although it was not as fast as Ilog. Unfortunately, it gave error messages when the problem size was increased, indicating that it does not scale well with respect to the number of FD variables. We have been able to solve larger problems by changing the size of certain environment variables but the performance

Table 3.11: Normalisation table for first solution search.

Benchmark	*Ilog*	*clp(fd)*	*Oz*	*CHR*	*SICStus*	*IF/Prolog*	*ECL^{i}PS^e*	*B-Prolog*
SRQ (10)	15.0	18.0	3.1	↓ 8.4	3.3	10.0	1.0	18.0
SRQ (50)	11.5	11.5	3.4	↓ 6.7	3.0	6.0	1.0	4.1
64 queens	98.3	4.00	2.6	-	1.6	2.7	1.0	54.1
100 queens	24.1	*Error1*	4.4	-	1.0	1.1	1.0	↓ 21.6
sendmore	3.3	1.7	1.7	-	2.0	2.0	1.0	3.3
alpha	12.6	5.4	3.4	-	1.8	↓ 67.4	1.0	7.0
eq. 10	4.2	3.1	1.8	-	1.2	5.5	1.0	2.3
eq. 20	4.4	2.9	2.3	-	1.4	7.0	1.0	1.3
magic(10)	16.7	8.3	3.3	-	1.5	1.5	1.0	1.2
magic(50)	74.9	7.6	4.4	-	2.3	2.2	1.0	1.56
magic(100)	177.5	11.5	3.1	-	3.6	2.4	1.0	*Error4*
magic(130)	216.0	*Error1*	2.5	-	4.3	2.7	1.0	*Error4*
magic(150)	87.1	*Error1*	↓ 1.1	-	1.6	1.0	*Error3*	*Error4*
magic(200)	195.0	*Error1*	*Error2*	-	3.2	1.0	*Error3*	*Error4*

was really poor. For example, with such a change, clp(FD) solved the **100 queens** problem for first solution search in 126 seconds (almost 23 times slower than ECL^{i}PS^e).

Oz. Oz was faster than ECL^{i}PS^e when finding the first and all solutions. When obtaining all solutions for the **magic sequences** problem, it was almost as fast as clp(FD). Also, Oz was more robust than clp(FD) and only failed to obtain a solution for the **magic sequences (200)** problem.

SICStus and **IF/Prolog**. These had very similar performance figures and were about two to three times as fast as ECL^{i}PS^e (although IF/Prolog performed badly with the **alpha** benchmark). Note that IF/Prolog worked particularly well for first solution search (even sometimes better than clp(FD) and Ilog). SICStus and IF/Prolog were also more robust than clp(FD), Oz, and ECL^{i}PS^e. However, of the two, SICStus has the greater robustness since it was able to solve the **magic sequences** problem with over 1000 FD variables whereas IF/Prolog failed to solve the same problem with 600 FD variables.

B-Prolog. As for clp(FD), this system worked well with problems involving a small number of FD variables. However, as we increased the number of FD variables for the **queens** and **magic sequences** problems, performance deteriorated rapidly leading, in most cases, to the program being aborted with error messages. This was a direct consequence of the fact that this version of B-Prolog does not have a garbage collector.

ECL^{i}PS^e. This had the slowest results (except for CHR). To obtain the best pos-

Table 3.12: Normalisation table for all solutions search

Benchmark	*Ilog*	*clp(fd)*	*Oz*	*SICStus*	*IF/Prolog*	*ECL^{i}PS^e*	*B-Prolog*
8 queens	6.4	5.9	1.0	1.4	4.5	1.0	10.9
9 queens	6.6	6.3	↓ 1.1	1.5	4.1	1.0	11.0
10 queens	6.4	6.5	↓ 1.1	1.5	4.2	1.0	10.7
11 queens	7.0	6.9	↓ 1.1	1.5	4.2	1.0	11.3
12 queens	7.1	7.0	1.0	1.5	4.2	1.0	11.6
magic(30)	31.6	5.7	5.7	2.6	3.0	1.0	↓ 1.2
magic(75)	43.0	6.4	5.7	3.9	3.6	1.0	↓ 1.8
magic(100)	46.7	6.8	5.1	4.7	3.7	1.0	*Error4*
magic(130)	11.1	*Error1*	1.2	1.4	1.0	*Error3*	*Error4*
magic(150)	11.5	*Error1*	1.1	1.5	1.0	*Error3*	*Error4*

sible performance, the figures in Table 3.7 for **ECL^{i}PS^e** had the garbage collection disabled. With garbage collection, the **magic sequences (100)** problem took three times longer to find a solution. We had no results and an error message for the scalable benchmarks when the number of FD variables was large. In fact, in many cases these problems could be solved but with a very poor performance. For example, without garbage collection, the **magic sequences (N)** problem with $\mathbf{N} \geq 130$ could not be solved. However, when the garbage collection was enabled, it was solved (on the Sparc Station) for first solution search in 480 seconds for $\mathbf{N} = 130$, 795 seconds for $\mathbf{N} = 150$ and 1977 seconds for $\mathbf{N} = 200$.

CHR. This was slowest. Two reasons for this: (1) the system we tested is built on top of ECL^{i}PS^e (which had poor performance) and (2) CHR was not designed primarily for efficiency but for defining adequate constraints solvers for particular problems on specific domains.

3.6 Related Work

Traditionally, the basic literature for each paradigm in computer science has a set of papers that compare different aspects of the paradigm itself as for example, complexity of algorithms, efficiency of systems, expressiveness of languages, applicability of resources to specific settings, etc. In particular, for CLP systems, there are a number of publications in which the constraint systems have been tested on the solving of a number of traditional benchmarks. In the following we discuss some of these works although the list is, of course, not exhaustive. Taking these chronologically:

- in (Cras, 1993), a review of the main constraint solving tools are presented. This book focuses on the commercial systems and is mainly directed at people working in the industry and does not add anything new (Jampel, 1994);

Table 3.13: Performance results on Sparc (25 MHz) for first solution search and *naive* labeling.

Benchmark	*Ilog*	*clp(FD)*	*Oz*	*CHR*	*ECL^{i}PS^e*	*B-Prolog*
SRQ (10)	0.060	0.050	0.290	7.600	0.890	0.050
SRQ (50)	0.120	0.120	0.405	9.300	1.383	0.350
sendmore	0.020	0.020	0.020	-	0.033	0.010
alpha	10.070	14.400	48.99	-	309.200	46.866
eq. 10	0.180	0.220	0.390	-	0.650	0.300
eq. 20	0.230	0.360	0.570	-	1.367	0.834
magic(10)	0.040	0.080	0.150	-	0.433	0.733
magic(50)	0.790	5.970	4.660	-	32.833	599.550
magic(100)	3.930	43.830	37.300	-	232.733	*Error4*
magic(130)	7.490	*Error1*	60.900	-	*Error3*	*Error4*
magic(150)	10.720	*Error1*	97.516	-	*Error3*	*Error4*
magic(200)	22.880	*Error1*	*Error2*		*Error3*	*Error4*

- (Sidebottom, 1993) describes a language based on projection constraints for compiling and optimising constraint propagation in the Boolean and numerical domains. This language is implemented in a CLP system called Nicolog that is compared with languages that process constraints using symbolic computations such as cc(FD) (Van Hentenryck et al., 1994), BNR(Prolog) (Older and Vellino, 1990), clp(FD) (Codognet and Diaz, 1996a), CLP(BNR) (Benhamou and Older, 1997), Echidna (Havens et al., 1992), Eristo (Erlt and Krall, 1992), CHIP (Dincbas et al., 1988b), CAL (Sakai and Aiba, 1989), CLP($\Re$) (Jaffar et al., 1992b) and Prolog III (Colmerauer, 1990). This comparison is mainly focused on the capabilities of Nicolog with respect to the rest of the compared systems. Efficiency is just compared with respect to the clp(FD) system;

- Codognet and Diaz have implemented several CLP systems and have written several papers comparing their systems on both Boolean and FD with other systems. In (Codognet and Diaz, 1996b) and (Codognet and Diaz, 1994) they described two specific propagation-based Boolean solvers called clp(B/FD) and clp(B) respectively. These systems were compared with the CHIP system as well as with some specific methods of resolution that accept a set of constraints as input and solve it. They also described the finite domain CLP system clp(FD) in (Codognet and Diaz, 1996a) and compared it again with the CHIP system;

- also, in (Puget and Leconte, 1995), there is a discussion about the kind of constructs needed to implement global constraints. A system was implemented and its efficiency compared with respect to local propagation implementations;

- in (Müller and Würtz, 1996), a C++ interface for the concurrent constraint language Oz for implementing non-basic constraints as propagators is described and its overall efficiency is compared with the systems clp(FD) and ECL^{i}PS^e;

Table 3.14: Performance results on PC (Linux) for first solution search and *naive* labeling.

Benchmark	*SICStus*	*IF/Prolog*	*ECL^{i}PS^e*
SRQ (10)	0.030	0.010	0.100
SRQ (50)	0.040	0.020	0.120
sendmore	0.005	0.005	0.010
alpha	5.930	51.280	23.360
eq. 10	0.035	0.010	0.045
eq. 20	0.060	0.010	0.090
magic(10)	0.020	0.020	0.040
magic(50)	0.710	0.540	2.180
magic(100)	2.910	3.170	14.190
magic(130)	5.130	6.480	31.400
magic(150)	7.150	9.140	*Error3*
magic(200)	13.350	25.410	*Error3*

- in (Carlsson et al., 1997), a finite domain constraint solver integrated in the SICStus system is described and its efficiency is compared with respect to a number of constraint systems.

Two main differences are found between our comparative work and most of the works referenced above. First, in contrast with our own comparative work, most comparative work in the literature of CLP, has been done by the language implementers themselves. Furthermore, the benchmarks used for the comparisons have been used in the development as well as the assessment of the languages so that such tests are biased. Our comparison involved no designer of any of the CLP languages studied so that it is more impartial than previous ones. Secondly, most of the existing comparative works were concerned solely with comparing the performance of specific characteristics of the involved systems. However, we also compared several aspects of the expressiveness of the languages.

3.7 Concluding Remarks

In this chapter, eight popular but very different constraint systems using a glass box approach have been compared on the Boolean and finite domains. Since, for some of these systems, the generating of new constraints requires a deeper knowledge of the implementation than for others, we have divided the systems into *black box* and *glass box* approaches. We have focused the comparison on the efficiency and on specific aspects of the expressivity (i.e., those concerning reified constraints and meta-constraints). By showing the main differences between the systems, we have provided some guidelines that should help the choice of an adequate constraint language for solving a specific (discrete) constraint satisfaction problem.

Table 3.15: Normalisation table for first solution search and *naive* labeling.

Benchmark	*Ilog*	*clp(fd)*	*Oz*	*CHR*	*SICStus*	*IF/Prolog*	*ECL^{i}PS^e*	*B-Prolog*
SRQ (10)	14.8	17.8	3.1	↓ 8.5	3.3	10.0	1.0	17.8
SRQ (50)	11.5	11.5	3.4	↓ 6.7	3.0	6.0	1.0	4.0
sendmore	1.7	1.7	1.7	-	2.0	2.0	1.0	3.4
alpha	30.7	21.5	6.3	-	3.9	↓ 2.2	1.0	6.6
eq. 10	3.6	3.0	1.7	-	1.3	4.5	1.0	2.2
eq. 20	5.9	3.8	2.4	-	1.5	9.0	1.0	1.6
magic(10)	10.8	5.4	2.9	-	2.0	2.0	1.0	↓ 1.7
magic(50)	41.6	5.5	7.0	-	3.1	4.0	1.0	↓ 18.3
magic(100)	59.2	5.3	6.2	-	4.9	4.5	1.0	*Error4*
magic(130)	52.3	*Error1*	6.4	-	6.1	4.8	1.0	*Error4*
magic(150)	10.6	*Error1*	1.2	-	1.3	1.0	*Error3*	*Error4*
magic(200)	13.8	*Error1*	*Error2*	-	1.9	1.0	*Error3*	*Error4*

Table 3.16: Number of FD variables managed in the **magic sequences** problem (1)

Ilog	*clp(fd)*	*Oz*	*ECL^{i}PS^e*	*B-Prolog*
1624-1625	111-112	194-200	230-???	82-83

It should be noted, that our conclusions are based on simple problems. Implementing these in the eight systems required a considerable amount of work and the tests needed a large amount of computing time and power. More resources are needed if more realistic problems are to be used in comparing and contrasting the CLP systems.

To summarise our results, for maximum efficiency Ilog SOLVER is best. clp(FD) is also a good candidate provided the size of the problem (measured in number of FD variables) is fairly small. From an expressive point of view, CHR is best. This supports the accepted view that CHR's are particularly useful for building specialised constraint solvers for non-standard applications. Finally, for a balance between the expressiveness and efficiency, IF/Prolog and SICStus both do well although, in the tests here, SICStus had the greater robustness.

3.8 Contributions

To our knowledge, this is the first time that such a number of constraint systems have been compared on the finite domains from both the expressiveness and the efficiency

Table 3.17: Number of FD variables managed in the **magic sequences** problem (2)

ECL^{i}PS^e	*SICStus*	*IF/Prolog*
228-???	1000-1200	300-600

point of view. We have compared not only the performance but also key aspects of the expressiveness of the languages. This chapter provides two main contributions:

- for the CLP community, the experience reported here can aid others in choosing an appropriate constraint language for solving their specific discrete constraint satisfaction problem.
- for this thesis, the results have helped justify the election of the indexical approach for defining a generic framework for solving interval constraints (as described in the following four chapters). Observe that, in general, the systems adopting the indexical approach provide a good balance between efficiency and expressiveness.

Part III

Theoretical Framework

Chapter 4

Interval Constraint Propagation on Lattice (Interval) Domains

[Boswell:] Sir, what is poetry?
[Johnson:] Why Sir, it is much easier to say what it is not.
We all know what light is
but it is not easy to tell what it is.

Boswell Life, vol.3, pag: 38 (1776)
Samuel Johnson, 1709-84

4.1 Motivations

As shown in Chapter 2, CLP systems support many different domains such as finite ranges of integers (Carlson et al., 1994a; Carlsson et al., 1997; Codognet and Diaz, 1996a), reals (Jaffar et al., 1992b; Refalo and Van Hentenryck, 1996; Sidebottom and Havens, 1992; Benhamou, 1995), finite sets (Walinsky, 1989; Gervet, 1997; Müller and Müller, 1997) or the Booleans (Codognet and Diaz, 1996b; Codognet and Diaz, 1994; Barth and Bockmayr, 1996). The type of the domain determines the nature of the constraints and the solvers used to solve them. In particular, the cardinality of the domain determines the constraint solving procedure so that existing CLP systems have distinct constraint solving methods for the finite and the infinite domains. However, in practice, constraint problems are often not specific to any particular domain and thus their formulation has to be artificially adapted to fit a given solver.

As also shown in Chapter 2, most constraint solvers, called *black box* solvers, have the control fixed by the system. This black box approach enables very efficient implementations and can provide practical tools for the common constraint applications. However, such black box solvers lack adaptability for use in solving non-standard problems. To overcome this lack of flexibility, many constraint systems provide *glass box*

solvers (Frühwirth, 1998; Codognet and Diaz, 1996a). These allow new constraints to be defined by the user. We are particularly interested in glass box systems that do not require the user to have a detailed knowledge of the implementation.

From this perspective (see Section 2.4), there have been two main separate developments for the provision of glass box constraints: the constraint system clp(FD) (Codognet and Diaz, 1996a) and the *Constraint Handling Rules (CHR)* (Frühwirth, 1998). The first of these, designed for the finite domain (FD) of integers, is based on a single constraint that allows the user to define and control the constraint propagation. This constraint, often referred to as *indexical*, is very efficient, as it was shown in Chapter 3, since its implementation uses a simple interval narrowing technique which can be smoothly integrated into the WAM (Aït-kaci, 1999; Diaz and Codognet, 1993). As a result, clp(FD) is now part of mainstream CLP systems such as SICStus Prolog (Carlsson et al., 1997) and IF/Prolog (If/Prolog, 1994). On the other hand, the CHR (now included as a library in SICStus Prolog (Carlsson et al., 1997)) enable the creation of new user-defined domains and their solvers and allow any interaction between them. Unfortunately the flexibility of these rules has an efficiency cost, as it was also shown in Chapter 3, and the CHR systems have not been able to compete with other systems that employ the more traditional approaches.

It follows from this discussion that what is needed is a glass box system that combines the flexibility of CHR with the efficiency of clp(FD). In the previous chapter we have shown that the systems supporting the indexical approach provided a good balance between expressiveness and efficiency. Based on this result, in this chapter we adopt the indexical approach of clp(FD) to constraint propagation, generalising it for any set of domains that are lattices, thereby providing a flexibility closer to that of CHR.

Observe that the framework, being applicable to any lattice, provides support for all the existing practical domains in CLP (e.g. reals, integers, sets and Booleans). Moreover, by using lattice combinators, new compound domains and their solvers can easily be obtained from previously defined domains such as these. Also, as the framework supports a set of lattice structure domains, it enables a form of constraint cooperation by means of one-way channels in which the information can flow between the domains.

In this chapter we provide a generic formalisation of the indexical approach for interval constraint propagation over domains with lattice structure that is new even in the case of the finite domains of integers where this approach is well-established and widely-used.

Our framework, being an alternative to the flexibility of CHR, has many advantages and, we believe, considerable potential as indicated below.

- The only condition we have placed on a domain is that it is a lattice. Since, as far as we know, all existing domains provided for CLP systems are already lattices or could be easily extended to become lattices, our framework supports a wide variety of applications.
- There are many well known lattice combinators such as the direct and lexicographic products so that it is straightforward to combine existing domains and

their solvers to form new (compound) domains.

- The framework is defined on a *set* of domains, allowing information to flow between domains so that distinct solvers on distinct domains can communicate and hence, cooperate.

- The basic schema for propagating constraints is uniform over all domains regardless of whether they are user-defined or system-defined and irrespective of their cardinality.

- Our framework is a totally transparent glass box setting over which new domains, new constraints on the computation domains and the intended propagation schema can be devised at the user level.

4.1.1 An Overview of Our Proposal

We conclude this motivation by giving an overview of our proposal which is described in more detail in the rest of this chapter. Consider a computation domain L such as, the integers or Booleans, then the solver applies interval reasoning to an ordered pair of elements of L. However, to allow for continuous and infinite domains, this underlying computation domains L is first replaced by two extended forms of the domain, a left and a right *bounded computation domain*. For the right bounded computation domain L^s each element of L is combined with one of two brackets '**)**' or '**]**'. Dually, for the left bounded computation domain $\overline{L^s}$, each element of $\hat{L}$ is combined with one of the brackets, '**(**' or '**[**'. Note that each element in L is represented by at least one element in L^s or $\overline{L^s}$. For example, consider the real domain (that is $L = \Re$), the real value 1.0 is represented by the value (1.0,'**]**') (resp. ($\hat{1.0}$,'**]**')) in $\Re^s$ (resp. $\overline{\Re^s}$) which, for clarity, is denoted as **1.0]** (resp. **[1.0**). Also, the greatest value smaller than 1.0 is represented by the value (1.0,'**)**') in $\Re^s$ and denoted as **1.0)**, whereas the lowest value higher than 1.0, denoted as **(1.0** is in $\overline{\Re^s}$. Operators such as + for the numeric domains and $\wedge$ in the Boolean domain are defined over the bounded domains by extending the definitions of + and $\wedge$ on the computation domain to include the brackets.

The interval domain which is used for the constraint propagation consists of the set of pairs $\langle a, b\rangle$, where a is in the left and b is in the right bounded computation domain and[1] $a \leq b$. The indexicals $max(x)$ and $min(x)$ are used to represent the greatest and least possible values for x in the bounded domains in which x is constrained. For example, the constraints

$$x \text{ in } \langle \mathbf{[4.0}, \mathbf{10.0]}\rangle, \quad x \neq \mathbf{10.0]},$$
$$y \text{ in } \langle \mathbf{[0.0}, max(x)\rangle$$

[1]Despite that a and b belong to different domains, in this chapter we show that they can be compared by applying the duality principle of lattices -see Section 4.2.

lead to the constraints

$$x \text{ in } \langle [\mathbf{4.0}, \mathbf{10.0}) \rangle,$$
$$y \text{ in } \langle [\mathbf{0.0}, \mathbf{10.0}) \rangle$$

since the first two constraints for x simply means that $max(x) = \mathbf{10.0})$.

4.1.2 Chapter Structure

This chapter is structured as follows. In Section 4.2, some fundamental algebraic concepts used in the rest of this document were recalled. Then, in Section 4.3, we define the computation domain and construct the interval domain used for interval constraint solving. In Section 4.4, the interval constraints are presented together with the procedure for constraint propagation and narrowing (called here, stabilisation). We also discuss the monotonicity of constraints in this section. In Section 4.5, we provide a schema for the operational semantics of our constraint solver and show how this can be adapted so as to ensure termination for infinite as well as finite domains. In Section 4.6, non-standard examples of computation domains are shown and also it is demonstrated how new domains can be constructed using different lattice combinators. The chapter ends with a discussion about related work, the conclusions and an enumeration of the major contributions of this chapter.

4.2 Preliminaries and Notation

If C is a set, then $\#C$ denotes its cardinality, $\wp(C)$ its power set and $\wp_f(C)$ the set of all the finite subsets of C, that is to say, $\wp_f(C) = \{c \in \wp(C) \mid c \text{ is finite}\}$.

Ordering. Let C be a set with equality. A binary relation $\preceq$ on C is an *ordering* relation if it is reflexive, antisymmetric and transitive. Let C be a set with ordering relation $\preceq$ and $c, c' \in C$. Then, write $c \preceq_C c'$ (when necessary) to express that $c \preceq c'$ and we write $c \sim c'$ if either $c \preceq c'$ or $c' \preceq c$ (i.e., c and c' are comparable) and $c \not\sim c'$ otherwise. Also $c \prec_C c'$ if $c \preceq_C c'$ and $c' \not\preceq_C c$. Any set C for which an ordering relation is defined, is said to be *ordered*. We say C is *totally ordered* if for any $a, b \in C$, $a \sim b$.

Bounds. Let C be an ordered set. An element c in C is a *lower (upper) bound* of a subset $E \subseteq C$ if and only if $\forall x \in E$: $c \preceq x$ ($x \preceq c$). If the set of lower (upper) bounds of E has a greatest (least) element, then that element is called the *greatest lower bound* (*least upper bound*) of E and denoted by $glb_C(E)$ ($lub_C(E)$). If $E = \{x, y\}$, we write $glb_C(x, y)$ to denote $glb_C(\{x, y\})$ and $lub_C(x, y)$ to denote $lub_C(\{x, y\})$.

Predecessor and successor. Let C be a totally ordered set and let $c, c' \in C$. Then c is the *immediate predecessor* of c' and c' the *immediate successor* of c if $c \prec c'$ and for any $c'' \in C$ with $c \preceq c'' \prec c'$ implies $c = c''$.

Monotonicity. Let f be a n-ary function $f :: C_1 \times \ldots \times C_n \to C$, where C and all C_i, for $i \in \{1, \ldots, n\}$, are ordered sets. Then we say that f is *monotonic in* C if, whenever $t_i, t_i' \in C_i$ such that $t_i \preceq_{C_i} t_i'$, for all $i \in \{1, \ldots, n\}$, then

$$f(t_1, \ldots, t_i, \ldots, t_n) \preceq_C f(t_1', \ldots, t_i', \ldots, t_n').$$

and *anti-monotonic* if (under the same conditions)

$$f(t_1, \ldots, t_i, \ldots, t_n) \succeq_C f(t_1', \ldots, t_i', \ldots, t_n').$$

A monotonic function f is *strict monotonic* if, whenever $t_i, t_i' \in C_i$ such that $t_i \preceq_{C_i} t_i'$, for all $i \in \{1, \ldots, n\}$ and $t_j \prec_{C_j} t_j'$, for some $j \in \{1, \ldots, n\}$, then

$$f(t_1, \ldots, t_i, \ldots, t_n) \prec_C f(t_1', \ldots, t_i', \ldots, t_n').$$

and *strict anti-monotonic* if (under the same conditions)

$$f(t_1, \ldots, t_i, \ldots, t_n) \succ_C f(t_1', \ldots, t_i', \ldots, t_n').$$

Lattice. Let L be an ordered set. L is a *lattice* if $lub_L(x, y)$ and $glb_L(x, y)$ exist, for any two elements $x, y \in L$.

Top and bottom elements. Let L be a lattice. If it exists, $glb_L(L) = \bot_L$ is *the bottom element* of L. Similarly, if it exists, $lub_L(L) = \top_L$ is *the top element* of L. The lack of a bottom or top element can be remedied by adding a fictitious one. Thus, the *lifted lattice* of L is $L \cup \{\bot_L, \top_L\}$ where, if $glb_L(L)$ does not exist, $\bot_L$ is a new element not in L such that $\forall a \in L, \bot_L \prec a$ and similarly, if $lub_L(L)$ does not exist, $\top_L$ is a new element not in L such that $\forall a \in L, a \prec \top_L$.

Dual. Let L be a lattice. The *dual of* L, denoted by $\hat{L}$, is the lattice that contains exactly the same elements as L and that is obtained by interchanging $glb_L(a, b)$ and $lub_L(a, b)$ for any $a, b \in L$. If $a \in L$, then we denote its dual as $\hat{a} \in \hat{L}$. The *duality principle for lattices* is "the dual of a statement about lattices phrased in terms of *glb* and *lub* can be obtained simply by interchanging *glb* and *lub*". Note that the ordering is also reversed in the dual lattice. That is, if $a, b \in L$ and $a \preceq_L b$, then $\hat{b} \preceq_{\hat{L}} \hat{a}$.

Products. Let L_1 and L_2 be two (lifted) lattices. Then the direct product $\langle L_1, L_2 \rangle$ and the lexicographic product (L_1, L_2) are lattices where:

$$\langle x_1, x_2 \rangle \preceq_{\langle L_1, L_2 \rangle} \langle y_1, y_2 \rangle \text{ iff } x_1 \preceq_{L_1} y_1 \text{ and } x_2 \preceq_{L_2} y_2;$$
$$(x_1, x_2) \preceq_{(L_1, L_2)} (y_1, y_2) \text{ iff } (x_1 \prec_{L_1} y_1) \text{ or } (x_1 = y_1 \text{ and } x_2 \preceq_{L_2} y_2).$$

Moreover,

$$\begin{aligned} glb(\langle x_1, x_2\rangle, \langle y_1, y_2\rangle) = &\quad \langle glb_{L_1}(x_1, y_1), glb_{L_2}(x_2, y_2)\rangle; \\ glb((x_1, x_2), (y_1, y_2)) = &\quad \text{if } x_1 = y_1 \text{ then } (x_1, glb_{L_2}(x_2, y_2)) \\ &\quad \text{elsif } x_1 \prec y_1 \text{ then } (x_1, x_2) \\ &\quad \text{elsif } x_1 \succ y_1 \text{ then } (y_1, y_2) \\ &\quad \text{else } (glb_{L_1}(x_1, y_1), \top_{L_2}); \end{aligned}$$

lub is the dual of *glb* and

$$\begin{aligned} \top_{\langle L_1,L_2\rangle} &= \langle \top_{L_1}, \top_{L_2}\rangle \text{ and } \bot_{\langle L_1,L_2\rangle} = \langle \bot_{L_1}, \bot_{L_2}\rangle; \\ \top_{(L_1,L_2)} &= (\top_{L_1}, \top_{L_2}) \text{ and } \bot_{(L_1,L_2)} = (\bot_{L_1}, \bot_{L_2}). \end{aligned}$$

It is straightforward to extend these definitions to n lattices.

Linear sum. Suppose that $L_1, \ldots, L_n$ are lattices. Then their *linear sum* $L_1 \uplus \ldots \uplus L_n$ is the lattice L_S where:
(1) $L_S = L_1 \cup \ldots \cup L_n$;
(2) the ordering relation $\preceq_{L_S}$ is defined:

$$x \preceq_{L_S} y \iff \begin{cases} x, y \in L_i \text{ and } x \preceq_{L_i} y; \text{ or} \\ x \in L_i, y \in L_j \text{ and } i \prec j; \end{cases}$$

(3) glb_{L_S} and lub_{L_S} are:

$$glb_{L_S}(x, y) = \left\{ \begin{array}{ll} glb_{L_i}(x, y) & \text{if } x, y \in L_i; \\ x & \text{if } x \in L_i,\ y \in L_j \text{ and } i \prec j; \\ y & \text{if } x \in L_i,\ y \in L_j \text{ and } j \prec i; \end{array} \right\}$$

$$lub_{L_S}(x, y) = \left\{ \begin{array}{ll} lub_{L_i}(x, y) & \text{if } x, y \in L_i; \\ y & \text{if } x \in L_i,\ y \in L_j \text{ and } i \prec j; \\ x & \text{if } x \in L_i,\ y \in L_j \text{ and } j \prec i; \end{array} \right\}.$$

and (4) $\bot_{L_S} = \bot_{L_1}$ and $\top_{L_S} = \top_{L_n}$.

For more information about lattices see, for example, (Davey and Priestley, 1990). In the rest of the document, $(L, \preceq_L, glb_L, lub_L, \bot_L, \top_L)$ denotes a (possible lifted) lattice on L with (possibly fictitious) bounds $\bot_L$ and $\top_L$.

4.3 The Computation and Interval Domains

The domain on which the values are actually computed, is called a *computation domain*. The key aspect of the constraint system described in this document is that it can be built on any computation domain provided it is a lattice. Throughout the document,

we let $\mathcal{L}$ denote a (possibly infinite) set of computation domains containing at least one element L and let $\hat{\mathcal{L}} = \{\hat{L} \mid L \in \mathcal{L}\}$. With each computation domain $L \in \mathcal{L}$, we associate a set of variable symbols V_L that is disjoint from L. We define $\mathcal{V}_{\mathcal{L}} = \cup\{V_L | L \in \mathcal{L}\}$. It is assumed (without loss of generality) that all $L \in \mathcal{L}$ are lifted lattices.

EXAMPLE 4.1 *Most classical constraint domains are lattices. For instance,*

$$(Integer, \leq, mini, maxi, \bot_{Integer}, \top_{Integer}),$$
$$(\Re, \leq, mini, maxi, \bot_{\Re}, \top_{\Re}),$$
$$(Bool, \leq, \wedge, \vee, false, true),$$
$$(Set\ L, \subseteq, \cap, \cup, \emptyset, L)$$

are lattices for the integers, reals, Booleans and sets, respectively, under their usual orders where mini and maxi functions return, respectively, the minimum and maximum element of any two elements in the integers or reals. Note that Integer and $\Re$ *are lifted lattices and include the fictitious elements* $\top_{Integer}$, $\bot_{Integer}$, $\top_{\Re}$ *and* $\bot_{\Re}$. *For the Booleans, it is assumed that* $Bool = \{false, true\}$ *with the ordering* $false <_{Bool} true$. *For the set lattice, we assume that* $Set\ \ L = \wp(L)$, *for each* $L \in \mathcal{L}$, *where* $\mathcal{L} = \{Integer, \Re, Bool\} \cup \{Set\ \ L \mid L \in \mathcal{L}\}$. *Note that* $\mathcal{L}$ *is an infinite set of computations domains.*

In the rest of the examples in this document, we will use the computation domains *Integer*, $\Re$, *Bool* and *Set* L for some domain $L \in \mathcal{L}$ without further comment or direct reference to this example.

Although the framework for our constraint system is based on the indexical approach of clp(FD) to constraint propagation which propagates constraints on finite closed intervals, this framework is intended for all lattices including infinite and continuous ones. For this reason, we need to be able to define the constraints over both open and closed intervals. Thus, in the rest of this section, we will show how a computation domain can be combined with a special binary lattice we call the *bracket domain* to form an *interval domain* suitable for the constraint propagation mechanism described in Section 4.4. We do this in a number of stages. First, in 4.3.1, we define the bracket domain and use this to construct two *bounded computational domains*, one for the left and the other for the right bound of an interval. We then extend these domains in 4.3.2 to allow for constraint operators. These are further extended in 4.3.3 to include an additional construct called an *indexical*. In 4.3.4, we use this enhanced bounded domain to construct the *interval domain*. Finally in 4.4.4 we introduce a simplification of the interval domain on discrete computation domains such as FD.

4.3.1 Bounded Computation Domains

We first define a domain of brackets,

DEFINITION 4.2 *(Bracket domain) The* bracket domain B *is a lattice containing just* ')' *and* ']' *with ordering* ')' $\prec_B$ ']'. *We let* '}' *denote any element of* B.

We also define the function $min_B(\}_1, \}_2)$ to return the least value of $\}_1$ and $\}_2$ in B.

This domain is combined with any computation domain to form the right bound of an interval.

DEFINITION 4.3 *(Simple bounded computation domain) The* simple bounded computation domain *(for L) is the lattice resulting from the lexicographic product (L, B) and is denoted L^s.*

We next define the *mirror* of this domain for the left bound of an interval.

DEFINITION 4.4 *(The mirror domain) The* mirror (of L^s) *is the lexicographic product $(\hat{L}, B)$ and is denoted by $\overline{L^s}$. The* mirror *of an element t=(a,'}') $\in L^s$ is the element $(\hat{a}, \text{'}\}\text{'}) \in \overline{L^s}$ and is denoted as $\overline{t}$.*

PROPOSITION 4.5

1. *$\overline{L^s}$ is the simple bounded computation domain for $\hat{L}$. i.e., $\overline{L^s} = \hat{L}^s$;*
2. *$\bot_{\overline{L^s}} = \overline{\top_L)} = (\top_L$ and $\top_{\overline{L^s}} = \overline{\bot_L]} = [\bot_L$.*
3. *If $t_1 = (a_1, \text{'}\}_1\text{'}), t_2 = (a_2, \text{'}\}_2\text{'}) \in L^s$ then,*

$$\text{if } a_1 \neq a_2, \left\{ \begin{array}{c} \overline{t_2} \prec_{\overline{L^s}} \overline{t_1} \iff t_1 \prec_{L^s} t_2 \\ glb_{\overline{L^s}}(\overline{t_1}, \overline{t_2}) = \overline{lub_{L^s}(t_1, t_2)} \\ lub_{\overline{L^s}}(\overline{t_1}, \overline{t_2}) = \overline{glb_{L^s}(t_1, t_2)} \end{array} \right\} \text{ else } \left\{ \begin{array}{c} \overline{t_1} \preceq_{\overline{L^s}} \overline{t_2} \iff t_1 \preceq_{L^s} t_2 \\ glb_{\overline{L^s}}(\overline{t_1}, \overline{t_2}) = \overline{glb_{L^s}(t_1, t_2)} \\ lub_{\overline{L^s}}(\overline{t_1}, \overline{t_2}) = \overline{lub_{L^s}(t_1, t_2)} \end{array} \right\}.$$

PROOF 4.6 *All three results are direct consequences of Definitions 4.3 and 4.4.* □

In the rest of the document, to simplify the notation, we often denote an element (a,'}') in L^s as **a}** and an element (â,'}') in $\overline{L^s}$ as **{a**. In particular, with this notation we have $\bot_{L^s} = \bot_L$), and $\top_{L^s} = \top_L$].

EXAMPLE 4.7 *Consider the computation domain L = Integer. Then* **6**] *denotes* (6, '] ') $\in$ *Integer^s,* [**6** *denotes* ($\hat{6}$, '] ') $\in \overline{Integer^s}$ *and* $\overline{\mathbf{6}]} = [\mathbf{6}$. *Also*

$$\begin{array}{l} glb_{L^s}(\mathbf{3}], \mathbf{5}]) = lub_{L^s}(\mathbf{3}], \mathbf{3})) = \mathbf{3}], \\ glb_{\overline{L^s}}([\mathbf{3}, [\mathbf{5}) = \overline{lub_{L^s}(\mathbf{3}], \mathbf{5}])} = \overline{\mathbf{5}]} = [\mathbf{5}, \\ lub_{\overline{L^s}}([\mathbf{3}, [\mathbf{5}) = \overline{glb_{L^s}(\mathbf{3}], \mathbf{5}])} = \overline{\mathbf{3}]} = [\mathbf{3}, \\ \mathbf{0}) \prec_{L^s} \mathbf{0}] \prec_{L^s} \mathbf{1}) \prec_{L^s} \mathbf{1}] \prec_{L^s} \ldots \prec_{L^s} \top_L) \prec_{L^s} \top_L] \text{ in } L^s, \\ (\top_L \prec_{\overline{L^s}} [\top_L \prec_{\overline{L^s}} \ldots \prec_{\overline{L^s}} (\mathbf{1} \prec_{\overline{L^s}} [\mathbf{1} \prec_{\overline{L^s}} (\mathbf{0} \prec_{\overline{L^s}} [\mathbf{0}. \end{array}$$

Moreover, if $L = Set\ Integer$, *then* $\{\mathbf{1},\mathbf{3}\})$ *denotes* $(\{1,3\}, \text{`}) \text{'})$, $(\{\mathbf{1},\mathbf{3}\}$ *denotes* $(\widehat{\{1,3\}}, \text{`}) \text{'})$ *and* $\overline{\{\mathbf{1},\mathbf{3}\}] = [\{\mathbf{1},\mathbf{3}\}}$. *Also*

$$
\begin{array}{l}
glb_{L^s}(\{\mathbf{4}\}], \{\mathbf{4},\mathbf{6}\}]) = \{\mathbf{4}\}] \ \ and\ lub_{L^s}(\{\mathbf{3}\}], \{\mathbf{4},\mathbf{6}\}]) = \{\mathbf{3},\mathbf{4},\mathbf{6}\}], \\
glb_{\overline{L^s}}([\{\mathbf{4}\}, [\{\mathbf{4},\mathbf{6}\}) = \overline{lub_{L^s}(\{\mathbf{4}\}], \{\mathbf{4},\mathbf{6}\}])} = \overline{\{\mathbf{4},\mathbf{6}\}]} = [\{\mathbf{4},\mathbf{6}\}, \\
lub_{\overline{L^s}}([\{\mathbf{4}\}, [\{\mathbf{4},\mathbf{6}\}) = \overline{glb_{L^s}(\{\mathbf{4}\}], \{\mathbf{4},\mathbf{6}\}])} = \overline{\{\mathbf{4}\}]} = [\{\mathbf{4}\}, \\
\{\mathbf{1}\}] \prec_{L^s} \{\mathbf{1},\mathbf{3}\}) \prec_{L^s} \{\mathbf{1},\mathbf{3}\}] \prec_{L^s} \{\mathbf{1},\mathbf{3},\mathbf{5}\}], \\
[\{\mathbf{1},\mathbf{3},\mathbf{5}\} \prec_{\overline{L^s}} (\{\mathbf{1},\mathbf{3}\} \prec_{\overline{L^s}} [\{\mathbf{1},\mathbf{3}\} \prec_{\overline{L^s}} [\{\mathbf{1}\}.
\end{array}
$$

PROPOSITION 4.8 *Let* $L' \in \{L^s, \overline{L^s}\}$ *and* $t \in L'$. *Then,*

$$(1)\ \overline{\overline{L'}} = L', \quad (2)\ \overline{L'} \in \{L^s, \overline{L^s}\}, \quad (3)\ \bar{t} \in \overline{L'} \quad \textit{and} \quad (4)\ \bar{\bar{t}} = t.$$

PROOF 4.9 *We prove the cases separately.*

(1) Observe that

$$\overline{\overline{L^s}} =^1 \overline{\overline{(L,B)}} =^2 (\hat{\hat{L}}, B) =^3 (L,B) =^4 L^s \tag{4.1}$$

where equalities, $=^1$ *and* $=^4$ *follow from Definition 4.3 on the facing page,* $=^2$ *from Definition 4.4 on the preceding page (applied twice), and* $=^3$ *from the duality definition for lattices.*

Thus, if $L' = L^s$, *then the result follows. Moreover, if* $L' = \overline{L^s}$, *then* $\overline{\overline{L'}} = \overline{\overline{\overline{L^s}}} = \overline{L^s} = L'$.

(2) By hypothesis, $\overline{L'} \in \{\overline{L^s}, \overline{\overline{L^s}}\}$. *Hence, by* (4.1), $\overline{L'} \in \{\overline{L^s}, L^s\}$.

(3) Observe that, as $t \in L'$, $t = (a, \text{`}\} \text{'})$, *for some* $a \in L \cup \hat{L}$, *we have*

$$\bar{t} =^1 \overline{(a, \text{`}\} \text{'})} =^2 (\hat{a}, \text{`}\} \text{'}) \in \overline{L'}$$

where the equality $=^1$ *follows from Definition 4.3, and equality* $=^2$ *and relation* $\in$ *from Definition 4.4.*

(4) Observe that, for some $a \in L \cup \hat{L}$, *we have*

$$\bar{\bar{t}} =^1 \overline{\overline{(a, \text{`}\} \text{'})}} =^2 (\hat{\hat{a}}, \text{`}\} \text{'}) =^3 (a, \text{`}\} \text{'}) =^4 t.$$

where equality $=^1$ *follows from Definition 4.3,* $=^2$ *from Definition 4.4 (applied twice) and* $=^3$ *from the duality definition for lattices.*

□

4.3.2 Constraint Operators

The bounded computation domains are extended to allow for operators over the computation domains.

DEFINITION 4.10 *(Constraint operators) Suppose* $L, L_1, \ldots, L_n \in \mathcal{L} \cup \hat{\mathcal{L}}$ *and that* $\circ_L$ *and* $\circ_B$ *are monotonic n-ary functions*

$$\circ_L :: L_1 \times \ldots \times L_n \to L,$$
$$\circ_B :: \underbrace{B \times \ldots \times B}_{n \text{ times}} \to B.$$

Suppose also that, if $\circ_L$ *is not a strict monotonic function then* $\circ_B$ *is a constant function. Let* $\circ$ *be the operator*

$$\circ :: L_1^s \times \ldots \times L_n^s \to L^s$$
$$\circ((a_1, \}_1), \ldots, (a_n, \}_n)) = (\ \circ_L(a_1, \ldots, a_n), \circ_B(\}_1, \ldots, \}_n)\ ). \tag{4.2}$$

Then $\circ$ *is called a* constraint operator for L^s.
The mirror $\overline{\circ}$ *of* $\circ$ *is*

$$\overline{\circ} :: \overline{L_1^s} \times \ldots \times \overline{L_n^s} \to \overline{L^s}$$
$$\overline{\circ}(\overline{t_1}, \ldots, \overline{t_i}, \ldots, \overline{t_n}) = \overline{\circ(t_1, \ldots, t_i, \ldots, t_n)}. \tag{4.3}$$

As is usual, if $\circ$ is a binary operator we allow for infix notation. Then the expression $\circ(t_1, t_2)$ will be denoted just as $t_1 \circ t_2$.

EXAMPLE 4.11 *Suppose* $+_L$ *and* $-_L$ *are declared as*

$$+_L :: L \times L \to L \qquad\qquad -_L :: L \times \hat{L} \to L.$$

This means that, when $L \in \{Integer, \Re\}$*, if the operators* $+_L/2$ *and* $-_L/2$ *have their usual definition and return the sum and difference of two numbers in L, they are strict monotonic. Observe that as a and* $\hat{b}$ *increase in L and* $\hat{L}$*, respectively,* $a -_L \hat{b}$ *also increases in L.*

Suppose, more generally, that, whenever $+_L/2$ *and* $-_L/2$ *are defined for a domain L, they are strict monotonic on L. Suppose also that* $+_B$ *and* $-_B$ *are defined as*

$$\}_1 +_B \}_2 = min_B(\}_1, \}_2) \qquad \text{and} \qquad \}_1 -_B \}_2 =]\ \text{if}\ \}_1 =_B \}_2\ \text{and}\ )\ \text{otherwise.}$$

Then the binary constraint operators $+$ *and* $-$ *for* L^s *and their mirrors are declared as*

$$+ :: L^s \times L^s \to L^s \qquad - :: L^s \times \overline{L^s} \to L^s$$
$$\overline{+} :: \overline{L^s} \times \overline{L^s} \to \overline{L^s} \qquad \overline{-} :: \overline{L^s} \times L^s \to \overline{L^s}.$$

Consider the case $L = \Re$. *Then we have*

$$\begin{aligned} &\mathbf{3.2}) + \mathbf{4.1}] = \mathbf{7.3}), && (\mathbf{3.2} \overline{+} [\mathbf{4.1} = \overline{\mathbf{3.2}) + \mathbf{4.1}]} = \overline{\mathbf{7.3})} = (\mathbf{7.3}, \\ &\mathbf{7.3}) - [\mathbf{4.1} = \mathbf{3.2}), && (\mathbf{7.3} \overline{-} \mathbf{4.1}] = \overline{\mathbf{7.3}) - [\mathbf{4.1}} = \overline{\mathbf{3.2})} = (\mathbf{3.2}. \end{aligned}$$

PROPOSITION 4.12 *Suppose* $\circ$ *is a constraint operator for* L^s. *Then,*

$$\circ \text{ is monotonic;} \tag{a}$$
$$\overline{\circ} \text{ is a constraint operator (for } \overline{L^s}\text{);} \tag{b}$$
$$\circ \text{ is the mirror of } \overline{\circ} \text{ i.e., } \circ \equiv \overline{\overline{\circ}}. \tag{c}$$

PROOF 4.13 *Suppose that* $\circ :: L_1^s \times \ldots \times L_n^s \to L^s$ *is a constraint operator. We prove the cases separately.*

(a) *Suppose* $t_i \preceq_{L_i^s} t_i'$ *for* $i \in \{1, \ldots, n\}$, *where* $t_i = (a_i, \}_i)$ *and* $t_i' = (a_i', \}_i')$. *We need to show*

$$\circ(t_1, \ldots, t_n) \preceq_{L^s} \circ(t_1', \ldots, t_n') \tag{4.4}$$

Observe that

$$\begin{aligned} &\text{By the product of lattices: } a_i \preceq_{L_i} a_i'; \text{ moreover if } a_i = a_i' \text{ then } \}_i \preceq_B \}_i'; \\ &\text{By monotonicity of } \circ_L : \qquad \circ_L(a_1, \ldots, a_n) \preceq_L \circ_L(a_1', \ldots, a_n'); \end{aligned}$$

If $\circ_L(a_1, \ldots, a_n) \prec_L \circ_L(a_1', \ldots, a_n')$ *then* (4.4) *holds by the product of lattices and Definition 4.10 on the facing page.*

Otherwise, $\circ_L(a_1, \ldots, a_n) = \circ_L(a_1', \ldots, a_n')$. *There are two cases:*

(1) $\forall i \in \{1, \ldots, n\}, a_i = a_i'$. *Then, by monotonicity of* $\circ_B$,

$$\circ_B(\}_1, \ldots, \}_n) \preceq_B \circ_B(\}_1', \ldots, \}_n').$$

(2) $\exists i \in \{1, \ldots, n\}, a_i \prec_{L_i} a_i'$. *Then* $\circ_L$ *is not a strict monotonic function and, by Definition 4.10,* $\circ_B$ *is a constant. i.e.,* $\circ_B(\}_1, \ldots, \}_n) = \circ_B(\}_1', \ldots, \}_n')$.

Thus, in both cases, (4.4) *holds by the product of lattices and Definition 4.10.*

(b) *Observe that*

$$\begin{aligned} &\overline{\circ} :: \overline{L_1^s} \times \ldots \times \overline{L_n^s} \to \overline{L^s}, && \text{(by Definition 4.10);} \\ &\overline{\circ} :: \hat{L_1}^s \times \ldots \times \hat{L_n}^s \to \hat{L}^s, && \text{(by Proposition 4.5(1));} \end{aligned}$$

where, for $i \in \{1, \ldots, n\}, \hat{L}_i \in \mathcal{L} \cup \hat{\mathcal{L}}$ *and* $\hat{L} \in \mathcal{L} \cup \hat{\mathcal{L}}$.

Assuming the notation of Definition 4.10, we have, if $t_i = (a_i, \}_i)$ $(1 \leq i \leq n)$, *then* $\circ(t_1, \ldots, t_n) = \big(\circ_L(a_1, \ldots, a_n), \circ_B(\}_1, \ldots, \}_n)\big)$.

Let $\circ_{\hat{L}} :: \hat{L_1} \times \ldots \times \hat{L_n} \to \hat{L}$ *be the dual operator to* $\circ_L$ *so that, if* $\circ_L(a_1, \ldots, a_n) = a$, *then* $\circ_{\hat{L}}(\hat{a_1}, \ldots, \hat{a_n}) = \hat{a}$. *Then, by the duality principle of lattices,* $\circ_{\hat{L}}$ *is monotonic in* $\hat{L}$. *Moreover,* $\circ_L$ *is strict monotonic whenever* $\circ_{\hat{L}}$ *is. Hence, we have*

$$\begin{aligned}
\overline{\circ}(\overline{t_1}, \ldots, \overline{t_n}) &= \overline{\circ(t_1, \ldots, t_n)} && \textit{(by Definition 4.10)} \\
&= \overline{(\ \circ_L(a_1, \ldots, a_n), \circ_B(\}_1, \ldots, \}_n)\ )} && \textit{(by Definition 4.10)} \\
&= (\ \circ_L(\widehat{a_1, \ldots}, a_n), \circ_B(\}_1, \ldots, \}_n)\ ) && \textit{(by Definition 4.4)} \\
&= (\ \circ_{\hat{L}}(\hat{a_1}, \ldots, \hat{a_n}), \circ_B(\}_1, \ldots, \}_n)\ ) && \textit{(by the definition of } \circ_{\hat{L}}\textit{)}.
\end{aligned}$$

Thus $\overline{\circ}$ *is a constraint operator as defined in Definition 4.10.*

(c) *Observe that,*

$$\begin{aligned}
\overline{\circ} &:: \overline{L_1^s} \times \ldots \times \overline{L_n^s} \to \overline{L^s}, && \textit{(by Definition 4.10);} \\
\overline{\overline{\circ}} &:: \overline{\overline{L_1^s}} \times \ldots \times \overline{\overline{L_n^s}} \to \overline{\overline{L^s}}, && \textit{(by Definition 4.10);} \\
\overline{\overline{\circ}} &:: L_1^s \times \ldots \times L_n^s \to L^s, && \textit{(by Proposition 4.8(1));}
\end{aligned}$$

Also if $t_i \in L_i^s$ *for all* $i \in \{1, \ldots, n\}$,

$$\begin{aligned}
\overline{\overline{\circ}}(t_1, \ldots, t_n) &= \overline{\overline{\circ}}(\overline{\overline{t_1}}, \ldots, \overline{\overline{t_n}}) && \textit{(by Proposition 4.8(4));} \\
&= \overline{\overline{\circ(t_1, \ldots, t_n)}} && \textit{(by Definition 4.10, applied twice);} \\
&= \circ(t_1, \ldots, t_n) && \textit{(by Proposition 4.8(4)).}
\end{aligned}$$

Therefore $\overline{\overline{\circ}}$ *is equivalent to* $\circ$.

□

In the definition of an operator, we allowed for the computation domains of its arguments to be different from the domain of the result. This can provide a communication channel from one solver to another.

EXAMPLE 4.14 *Suppose that the unary operator trunc is defined as*

$$\begin{aligned}
&\mathit{trunc} :: \Re^s \to \mathit{Integer}^s \\
&\mathit{trunc}(\boldsymbol{a}\}) = \mathit{trunc}_{\mathit{Integer}}(\boldsymbol{a})\ \mathit{trunc}_B(\})
\end{aligned}$$

where $\mathit{trunc}_{\mathit{Integer}}(a)$ *is defined to return the integer part of* a, *for any* $a \in \Re$, *and* $\mathit{trunc}_B(\}) =]$ *for any* $\} \in B$. *Then*

$$\overline{\mathit{trunc}} :: \overline{\Re^s} \to \overline{\mathit{Integer}^s}.$$

Thus, by (4.2) and (4.3),

$$\begin{aligned}
\mathit{trunc}(\mathbf{3.1}]) &= \mathit{trunc}_{\mathit{Integer}}(3.1)\mathit{trunc}_B(]) = \mathbf{3}], \\
\overline{\mathit{trunc}}([\mathbf{3.1}) &= \overline{\mathit{trunc}(\mathbf{3.1}])} = \overline{\mathbf{3}]} = [\mathbf{3}.
\end{aligned}$$

4.3.3 Indexicals

The variables in V_L are introduced into the domain L^s and its mirror by means of *indexicals*. Let O_L be a set of constraint operators defined for L^s.

DEFINITION 4.15 *(Bounded computation domain) The* bounded computation domain L^b *(for L) and its mirror $\overline{L^b}$ are defined:*

$$\begin{aligned} L^b &= L^s \\ &\quad \cup \{max(x) \mid x \in V_L\} \\ &\quad \cup \{val(x) \mid x \in V_L\} \\ &\quad \cup \left\{ \circ(t_1, \ldots, t_n) \;\middle|\; \begin{array}{l} \circ :: L'_1 \times \ldots \times L'_n \to L^s \in O_L, \\ t_i \in (L'_i)^b \text{ for } (1 \leq i \leq n) \end{array} \right\}, \\ \overline{L^b} &= \{\overline{t} \mid t \in L^b\} \end{aligned}$$

where for $i \in \{1, \ldots, n\}$, $L_i \in \mathcal{L}$ and $L'_i \in \{L^s_i, \overline{L^s_i}\}$,

$(L'_i)^b = L^b_i$ *if* $L'_i = L^s_i$ *and* $(L'_i)^b = \overline{L^b_i}$ *if* $L'_i = \overline{L^s_i}$*;*

$\overline{max(x)} = min(x)$, $\overline{min(x)} = max(x)$ *and* $\overline{\overline{val(x)}} = val(x)$*;*

for each $\circ :: L'_1 \times \ldots \times L'_n \to L^s \in O_L$,

$$\overline{\circ(t_1, \ldots, t_i, \ldots, t_n)} = \overline{\circ}(\overline{t_1}, \ldots, \overline{t_i}, \ldots, \overline{t_n}).$$

The expressions $max(x), min(x), val(x)$ and $\overline{val(x)}$ are called indexicals. *Elements of L^b and $\overline{L^b}$ are called* indexical terms.

The bounded computation domain L^b is also a lattice inheriting its ordering from L^s and where $\top_{L^b} = \top_{L^s}$ and $\bot_{L^b} = \bot_{L^s}$. Thus, if $t_1, t_2 \in L^b$, then $t_1 \preceq_{L^b} t_2$ if and only if $t_1 = \bot_{L^b}$, $t_2 = \top_{L^b}$ or $t_1, t_2 \in L^s \setminus \{\bot_{L^s}, \top_{L^s}\}$ and $t_1 \preceq_{L^s} t_2$.

PROPOSITION 4.16

(1) $\overline{\overline{t}} = t$ *for any* $t \in \{L^b, \overline{L^b}\}$.
(2) *If* $t \in L^b$ *then* $\overline{t} \in \overline{L^b}$ *and if* $t \in \overline{L^b}$ *then* $\overline{t} \in L^b$.

PROOF 4.17 *We prove each case separately.*

(1) Let $n(t)$ be the number of operators of t. We prove it by induction on $n(t)$.

- Base case*: $n(t) = 0$.*
  - *If $t \in \{L^s, \overline{L^s}\}$, then $\overline{\overline{t}} = t$ by Proposition 4.8(4).*
  - *If $t \in \{max(x), val(x), min(x), \overline{val(x)}\}$ for any $x \in V_L$ and $L \in \mathcal{L}$, then $\overline{\overline{t}} = t$ by Definition 4.15.*

- Non-base case: $n(t) > 0$.
  Let $t = \circ(t_1, \ldots, t_n) \in L^b \cup \overline{L^b}$. *Then* $n(t_1), \ldots, n(t_n) < n(t)$ *so that the inductive hypothesis can be assumed for* $t_1, \ldots, t_n$. *Therefore,*

$$\begin{aligned}
\bar{\bar{t}} &= \overline{\overline{\circ(t_1, \ldots, t_n)}} \\
&= \bar{\bar{\circ}}(\overline{\overline{t_1}}, \ldots, \overline{\overline{t_n}}), && \text{by Definition 4.15, applied twice;} \\
&= \ \circ\,(\overline{\overline{t_1}}, \ldots, \overline{\overline{t_n}}), && \text{by Proposition 4.12(c);} \\
&= \ \circ\,(t_1, \ldots, t_n) = t. && \text{by the inductive hypothesis.}
\end{aligned}$$

(2) If $t \in L^b$ *then* $\bar{t} \in \overline{L^b}$ *by Definition 4.15. If* $t \in \overline{L^b}$ *then, there are three cases:*

$$\begin{aligned}
&(a)\ t \in \overline{L^s} \Rightarrow \bar{t} \in \overline{\overline{L^s}} \Rightarrow^1 \bar{t} \in L^s \Rightarrow^2 \bar{t} \in L^b; \\
&(b)\ t \in \{min(x), \overline{val(x)}\} \Rightarrow^2 \bar{t} \in L^b; \\
&(c)\ t = \overline{\circ(t_1, \ldots, t_n)} \in L^b \Rightarrow^{2,3} \bar{t} \in L^b,
\end{aligned}$$

where $\Rightarrow^1$ *follows from Proposition 4.8(1),* $\Rightarrow^2$ *follows from Definition 4.15 and* $\Rightarrow^3$ *follows Proposition 4.16(1).*

□

EXAMPLE 4.18 *Let* $+$ *(for* $L = Integer$*),* $-$ *(for* $L = \Re$*) and trunc as defined in Examples 4.11 and 4.14. Then* $+, trunc \in O_{Integer}$, $- \in O_{\Re}$. *Let also* $x \in V_{Integer}$ *and* $y \in V_{\Re}$, *then:*

$\mathbf{3}]$,	$max(x)$,	$\mathbf{3}) + max(x)$,	$trunc(max(y))$	*are in* $Integer^b$;
$\mathbf{20.1})$,	$max(y)$,	$\mathbf{20.1}) - min(y)$,	$val(y)$	*are in* $\Re^b$;
$[\mathbf{3}$,	$min(x)$,	$(\mathbf{3}\overline{+}min(x)$,	$\overline{trunc}(min(y))$	*are in* $\overline{Integer^b}$;
$(\mathbf{20.1}$,	$min(y)$,	$(\mathbf{20.1}\overline{-}max(y)$,	$\overline{val(y)}$	*are in* $\overline{\Re^b}$

where, for instance, $\overline{\mathbf{3}) + max(x)} = \overline{\mathbf{3})}\overline{+}\overline{max(x)} = (\mathbf{3}\overline{+}min(x)$.

4.3.4 Interval Domains

We now define the structure over which constraints will be solved.

DEFINITION 4.19 *(Interval domain) Let* L^b *be the bounded computation domain for a domain* $L \in \mathcal{L}$. *The* interval domain R_L^b *over* L *is the direct product* $\langle \overline{L^b}, L^b \rangle$. *The* simple interval domain R_L^s *over* L *is the direct product* $\langle \overline{L^s}, L^s \rangle$. *A* range *is an element of* R_L^b. *It is* simple *if it is an element of* R_L^s.

By the product of lattices, R_L^b and R_L^s are lattices. To simplify the notation, an element $\langle \overline{\mathbf{a}\}}, \mathbf{b}\}\rangle$ is written as **{a,b}**.

Therefore, for any $r_1 = \langle \overline{s_1}, t_1\rangle, r_2 = \langle \overline{s_2}, t_2\rangle \in R_L^b$ where $s_1, s_2, t_1, t_2 \in L^b$,

$$
\begin{aligned}
r_1 \preceq_{R_L^b} r_2 &\iff (\overline{s_1} \preceq_{\overline{L^b}} \overline{s_2}) \text{ and } (t_1 \preceq_{L^b} t_2),\\
glb_{R_L^b}(r_1, r_2) &= \langle glb_{\overline{L^b}}(\overline{s_1}, \overline{s_2}), glb_{L^b}(t_1, t_2)\rangle,\\
lub_{R_L^b}(r_1, r_2) &= \langle lub_{\overline{L^b}}(\overline{s_1}, \overline{s_2}), lub_{L^b}(t_1, t_2)\rangle,\\
\top_{R_L^b} &= [\bot_L, \top_L] \text{ and } \bot_{R_L^b} = (\top_L, \bot_L).
\end{aligned}
$$

EXAMPLE 4.20

$$
\begin{gathered}
\langle \overline{\mathbf{1}], \mathbf{8})}\rangle \in R^s_{Integer} \textit{ is written as } [\mathbf{1}, \mathbf{8}),\\
[\mathbf{2.3}, \mathbf{8.9}) \in R^s_{\Re} \quad \textit{and} \quad [\mathbf{1.4}, max(x)+\mathbf{4.9}] \in R^b_{\Re},\\
glb_{R^b_{\Re}}([\mathbf{3.2}, \mathbf{6.7}], (\mathbf{1.8}, \mathbf{4.5}]) = [\mathbf{3.2}, \mathbf{4.5}],\\
lub_{R^b_{\Re}}([\mathbf{3.2}, \mathbf{6.7}], (\mathbf{1.8}, \mathbf{4.5}]) = (\mathbf{1.8}, \mathbf{6.7}],\\
[\mathbf{3.0}, \mathbf{4.0}) \preceq_{R^b_{\Re}} (\mathbf{1.8}, \mathbf{4.5}].
\end{gathered}
$$

Note that

$$
\begin{aligned}
glb_{R^b_{\Re}}([\mathbf{3.2}, \mathbf{6.7}], (\mathbf{1.8}, \mathbf{4.5}]) &= \langle\ glb_{\overline{\Re^b}}(\ [\mathbf{3.2}, (\mathbf{1.8}\ ), glb_{\Re^b}(\ \mathbf{6.7}], \mathbf{4.5}]\ )\ \rangle = [\mathbf{3.2}, \mathbf{4.5}];\\
lub_{R^b_{\Re}}([\mathbf{3.2}, \mathbf{6.7}], (\mathbf{1.8}, \mathbf{4.5}]) &= \langle\ lub_{\overline{\Re^b}}(\ [\mathbf{3.2}, (\mathbf{1.8}\ ), lub_{\Re^b}(\ \mathbf{6.7}], \mathbf{4.5}]\ )\ \rangle = (\mathbf{1.8}, \mathbf{6.7}].
\end{aligned}
$$

It is important to note that $\preceq_{R_L^b}$ simulates the interval inclusion[2]. Figure 4.1 illustrates the relationship between any two elements of the interval domain R_L^s over any computation domain $L \in \mathcal{L}$.

The consistency of simple ranges can be generically defined in any R_L^s as follows.

DEFINITION 4.21 *(Consistency conditions) A simple range* $r = \langle \overline{s}, t\rangle \in R_L^s$ *is* inconsistent *if*

1. $s \not\preceq_{L^s} t$ *(note that this means that* r *is inconsistent if* $s \not\prec_{L^s} t$*) or*
2. $\overline{s} = (\boldsymbol{a}$ *and* $t = \boldsymbol{a}\}$ *for any* $\} \in B$.

Otherwise r *is* consistent. *Note that this means that* $\bot_{R_L^b} = \bot_{R_L^s} = \langle \bot_{\overline{L^s}}, \bot_{L^s}\rangle = (\top_L, \bot_L)$ *is inconsistent.*

EXAMPLE 4.22 *In the domain Integer,* $(\mathbf{1}, \mathbf{1}]$ *and* $[\mathbf{5}, \mathbf{2}]$ *are inconsistent whereas* $[\mathbf{1}, \mathbf{10}]$ *is consistent. In the domain Set Integer,* $[\{\mathbf{1}, \mathbf{3}\}, \{\mathbf{1}\}]$ *and* $[\{\mathbf{1}, \mathbf{3}\}, \{\mathbf{1}, \mathbf{4}\}]$ *are inconsistent and* $[\{\mathbf{1}\}, \{\mathbf{1}, \mathbf{3}\}]$ *is consistent, since* $\{\mathbf{1}\}] \prec_{(Set\ Integer)^s} \{\mathbf{1}, \mathbf{3}\}]$.

[2]Note that the concept of interval used here differs from the usual concept of interval in set theory.

Figure 4.1: Relationship between any two ranges $\overline{s_1}, t_1$ and $\overline{s_2}, t_2$ in R^s_L for some $L \in \mathcal{L}$

Any (in)consistent range in R^s_L identifies a set of (in)consistent ranges in R^s_L.

PROPOSITION 4.23 *Suppose $r, r' \in R^s_L$, for any $L \in \mathcal{L}$, where $r \preceq_{R^s_L} r'$. If r' is inconsistent, then r is also inconsistent.*

PROOF 4.24 *Suppose that*

$$\begin{array}{rr} r = \langle \overline{s}, t \rangle & r' = \langle \overline{s'}, t' \rangle \\ s = (a, \}_1) & s' = (a', \}'_1) \\ t = (b, \}_2) & t' = (b', \}'_2). \end{array}$$

By hypothesis, $r \preceq_{R^s_L} r'$ and, by Definition 4.19 on page 92 and by the product of lattices (i.e., direct product),

$$\overline{s} \preceq_{\overline{L^s}} \overline{s'}; \tag{4.5}$$

$$t \preceq_{L^s} t'. \tag{4.6}$$

From (4.5)

$$\begin{aligned} \overline{s} \preceq_{\overline{L^s}} \overline{s'} &\Rightarrow^1 (\hat{a}, \}_1) \preceq_{(\hat{L},B)} (\hat{a'}, \}'_1) \\ &\Rightarrow^2 \hat{a} \prec_{\hat{L}} \hat{a'} \text{ or } \hat{a} = \hat{a'} \text{ and } \}_1 \preceq_B \}'_1 \\ &\Rightarrow^3 a' \prec_L a \text{ or } a = a' \text{ and } \}_1 \preceq_B \}'_1 \\ &\Rightarrow^2 \begin{cases} \text{if } a \prec_L a' \text{ then } s' \prec_{L^s} s; \\ \text{if } a = a' \text{ then } s \preceq_{L^s} s'. \end{cases} \end{aligned} \tag{4.7}$$

where $\Rightarrow^1$ *comes from Definition 4.4 on page 86, where* $\Rightarrow^2$ *comes from the product of lattices (i.e., the lexicographic product) and* $\Rightarrow^3$ *comes from the duality principle for lattices (see Section 4.2).*

We suppose that r' *is inconsistent. Then, by Definition 4.21 on page 93, we have three cases:*

$$t' \prec_{L^s} s'; \tag{i}$$
$$s' \not\prec t'; \tag{ii}$$
$$s' = a') \text{ and } t' = a\}'_2. \tag{iii}$$

We reason with respect to two cases: $a \prec_L a'$ *and* $a = a'$*. In the following*

$\Rightarrow^4$ *follows from* (4.7)*;*

$\Rightarrow^5$ *follows from* (4.6)*;*

$\Rightarrow^6$ *follows from Definition 4.21;*

$\Rightarrow^7$ *follows from Definition 4.3 on page 86;*

$\Rightarrow^8$ *follows from the product of lattices (i.e., the lexicographic product);*

$\Rightarrow^9$ *follows from a contradiction.*

Suppose $a \prec_L a'$*. Then,*

for (i), $a \prec_L a' \Rightarrow^5 t \prec_{L^s} s' \Rightarrow^4 t \prec_{L^s} s \Rightarrow^6 r$ *is inconsistent*;

for (ii), $a \prec_L a' \Rightarrow^{4,5} s' \prec_{L^s} s$ *and* $t \preceq_{L^s} t' \Rightarrow^* t \not\preceq_{L^s} s \Rightarrow^6 r$ *is inconsistent*;

for (iii), $a \prec_L a' \Rightarrow^8 s \prec_{L^s} s' \Rightarrow^4$ *false*.

where $\Rightarrow^*$ *follows from the fact that if* $t \succeq_{L^s} s$*, then* $s' \prec_{L^s} s \preceq_{L^s} t \preceq_{L^s} t'$ *which contradicts the hypothesis that* $s' \not\prec t'$*.*

Suppose now that $a = a'$*. Then,*

for (i),

$$a = a' \Rightarrow^4 s \preceq_{L^s} s' \Rightarrow \begin{cases} s = s' \Rightarrow^{10} t' \prec_{L^s} s \Rightarrow^5 t \prec_{L^s} s \Rightarrow^6 r \text{ is inconsistent}; \\ s \prec_{L^s} s' \Rightarrow^7 s = a) \text{ and } s' = a] \Rightarrow^{10} t' \prec_{L^s} a] \\ \qquad \Rightarrow^8 t' \preceq_{L^s} a) \Rightarrow^5 t \preceq_{L^s} a) \Rightarrow^6 r \text{ is inconsistent}. \end{cases}$$

for (ii),

$$a = a' \Rightarrow^{11} \begin{cases} b = b' \Rightarrow^{12} a \not\prec_L b \Rightarrow^8 s \not\prec t \Rightarrow^6 r \text{ is inconsistent}; \\ b \prec_L b' \text{ and} \begin{cases} b \succ_L a \Rightarrow^8 t \succ_{L^s} s' \Rightarrow^5 s' \preceq_{L^s} t' \Rightarrow^9 \text{false}; \\ b \prec_L a \Rightarrow^8 t \prec_{L^s} s \Rightarrow^6 r \text{ is inconsistent}; \\ b = a \Rightarrow a \prec_L b' \Rightarrow a' \prec_L b' \Rightarrow^8 s' \prec_{L^s} t' \Rightarrow^9 \text{false}. \end{cases} \end{cases}$$

for (iii),

$$a = a' \Rightarrow^4 s = a') \Rightarrow^5, \begin{cases} t = t' \Rightarrow t = a'\}'_2 \Rightarrow^6 r \text{ is inconsistent}; \\ t \prec_{L^s} t' \begin{cases} \Rightarrow^8 b = a' \text{ and } \}_2 \preceq_B \}'_2 \Rightarrow t = a'\}_2 \\ \qquad \Rightarrow^6 r \text{ is inconsistent}; \\ \Rightarrow^8 b \prec_L a' \Rightarrow^8 t \prec_L s \Rightarrow^6 r \text{ is inconsistent}. \end{cases} \end{cases}$$

where

$\Rightarrow^{10}$ *comes from case* (i) *(i.e.,* $t' \prec_{L^s} s'$*);*

$\Rightarrow^{11}$ *comes from* (4.6) *since* $t \preceq_{L^s} t'$ *and, by the product of lattices (i.e., lexicographic product),* $b \preceq_L b'$*;*

$\Rightarrow^{12}$ *comes from case* (ii) *so that* $s' \not\sim t'$*. Then, by Definition 4.3 and by the product of lattices (i.e., lexicographic product)* $a' \not\sim_L b'$ *and thus* $a \not\sim_L b'$*.*

Thus, in all cases, r is inconsistent.
□

EXAMPLE 4.25 *Suppose* $L \in \mathcal{L}$ *and* $a, b, c \in L$ *where* $a \prec_L c \prec_L b$*. Figure 4.2 illustrates the part of lattice* R_L^s *constructed from the elements* a, b *and* c*. Note that the nodes within the square are all inconsistent ranges where the rest of nodes are all consistent. The nodes with circles are special cases and are considered in Section 4.4.4.*

4.4 The Constraint Domains

The constraints are expressed by coupling each variable with one or more intervals of values (defined using the interval domain) over which the variable may range. These *interval constraints* which are defined in 4.4.1 are then solved using two processes: *constraint stabilisation* (i.e., constraint narrowing) and *constraint propagation* defined, respectively, in 4.4.2 and 4.4.3. In 4.4.5, we provide the concept of solution to a set of constraints. Finally, in 4.4.6, we explain how our constraint system ensures the monotonic propagation of constraints and thus ensures the reducing of the sizes of the intervals in the constraint propagation process.

We continue to use L to denote any domain in $\mathcal{L}$, V_L the set of variables associated with L. We also let R_L^b be the interval domain over L, $\mathcal{V}_\mathcal{L} = \cup\{V_L | L \in \mathcal{L}\}$ and $X \in \wp_f(\mathcal{V}_\mathcal{L})$.

4.4.1 Interval Constraints

The *interval constraints* assign elements of R_L^b to variables in V_L.

DEFINITION 4.26 *(Interval constraint domain) Let* $x \in V_L$ *and* $r \in R_L^b$*. Then*

$$x \sqsubseteq r$$

is called an interval constraint *for* L *with* constrained variable x*. If* r *is a simple range (resp. non-simple range), then* $x \sqsubseteq r$ *is called a* simple interval constraint *(resp.* non-simple interval constraint*). If* r *is consistent (resp. inconsistent), then* $x \sqsubseteq r$ *is* consistent *(resp.* inconsistent*). If* $r = \top_{R_L^b}$*, then* $x \sqsubseteq r$ *is called a* type constraint for x *and denoted by* $x ::' L$*. If* $t \in L$*, then* $x = t$ *is a shorthand for* $x \sqsubseteq [\mathbf{t}, \mathbf{t}]$*. The*

Figure 4.2: Structure of the simple interval domain R_L^s where $a, b, c \in L$ and $a \prec_L c \prec_L b$

interval constraints domain over X for L *is the set of all interval constraints for* L *with constrained variables in* X *and is denoted by* $\mathcal{C}_L^X$. *The union*

$$\mathcal{C}^X \stackrel{\text{def}}{=} \bigcup \{\mathcal{C}_L^X \mid L \in \mathcal{L}\}$$

is called the interval constraint domain over X *for* $\mathcal{L}$.

The ordering for $\mathcal{C}^X$ *is inherited from the ordering in* R_L^b. *We define* $c_1 \preceq_{\mathcal{C}^X} c_2$ *if and only if, for some* $L \in \mathcal{L}$, $c_1 = x \sqsubseteq r_1, c_2 = x \sqsubseteq r_2 \in \mathcal{C}_L^X$ *and* $r_1 \preceq_{R_L^b} r_2$.

In the rest of the document $x_1, \ldots, x_n ::' L$ denotes $x_1 ::' L, \ldots, x_n ::' L$.

EXAMPLE 4.27 *Examples of type constraints, simple constraints and non-simple interval constraints are shown below in (a), (b) and (c), respectively, where + (for* $L = \Re$*), − (for* $L = Integer$*) and trunc, are as defined in Examples 4.11 and 4.14.*

$$\begin{array}{ll}
(a) & x, y ::' Integer, \qquad b ::' Bool, \qquad w, r ::' \Re; \\
(b) & y \sqsubseteq [\mathbf{1}, \mathbf{4}), \qquad b \sqsubseteq [true, true], \qquad r \sqsubseteq [\mathbf{1.23}, \mathbf{3.45}); \\
(c) & r \sqsubseteq min(w), max(w) + \mathbf{3.2}] \quad and \\
 & x \sqsubseteq \overline{trunc}(min(w)) \overline{-} max(y), trunc(max(w)) - min(y).
\end{array}$$

DEFINITION 4.28 *(Intersection of simple interval constraints) Suppose* $x \in X$. *The intersection in a domain* $L \in \mathcal{L}$ *of two simple constraints* $c_1, c_2 \in \mathcal{C}_L^X$ *where* $c_1 = x \sqsubseteq r_1$, $c_2 = x \sqsubseteq r_2$ *and* $x \in V_L$ *is defined as follows:*

$$c_1 \cap_L c_2 = glb_{\mathcal{C}_L^X}(c_1, c_2) = x \sqsubseteq glb_{R_L^s}(r_1, r_2).$$

Suppose $x \in X$ and $c_1, c_2, c_3 \in \mathcal{C}_L^X$ are simple interval constraints with constrained variable x where $c_3 = c_1 \cap_L c_2$. Then it follows, from the definition, that $\cap_L$ has the properties:

Contractance: $c_3 \preceq_{\mathcal{C}_L^X} c_1$ and $c_3 \preceq_{\mathcal{C}_L^X} c_2$;

Correctness: if $c \preceq_{\mathcal{C}_L^X} c_1$ and $c \preceq_{\mathcal{C}_L^X} c_2$, then $c \preceq_{\mathcal{C}_L^X} c_3$;

Commutativity: $(c_1 \cap_L c_2) = (c_2 \cap_L c_1)$;

Idempotence: $(c_1 \cap_L c_3) = c_3$ and $(c_3 \cap_L c_2) = c_3$.

If S_x is a set of simple constraints for a variable $x \in V_L$, for some $L \in \mathcal{L}$, then we define $\bigcap_L S_x = glb_{\mathcal{C}_L^X}(S_x)$. As a result of the contractance property we have that $\bigcap_L S_x \preceq c$, for each $c \in S_x$.

EXAMPLE 4.29 *Consider the domains* $Integer, \Re, Bool$ *and* $Set\ Integer$*. Let* $X = \{i, r, b, s\}$ *where* $i \in V_{Integer}$*,* $r \in V_{\Re}$*,* $b \in V_{Bool}$ *and* $s \in V_{Set\ Integer}$*. Then*

$$\begin{aligned} i \sqsubseteq [\mathbf{5}, \mathbf{24}] \ \cap_L \ i \sqsubseteq [\mathbf{1}, \mathbf{15}) &= glb_{\mathcal{C}^X_{Integer}}\big(i \sqsubseteq [\mathbf{5}, \mathbf{24}], i \sqsubseteq [\mathbf{1}, \mathbf{15})\big) \\ &= i \sqsubseteq glb_{R^b_{Integer}}\big([\mathbf{5}, \mathbf{24}], [\mathbf{1}, \mathbf{15})\big) \\ &= i \sqsubseteq glb_{\overline{Integer^b}}\big([\mathbf{5}, [\mathbf{1}), glb_{Integer^b}(\mathbf{24}], \mathbf{15})\big) \\ &= i \sqsubseteq [\mathbf{5}, \mathbf{15}). \end{aligned}$$

Also,

$$\begin{aligned} r \sqsubseteq [\mathbf{1.12}, \mathbf{5.67}) \ \cap_L \ r \sqsubseteq [\mathbf{2.34}, \mathbf{5.95}) &= r \sqsubseteq [\mathbf{2.34}, \mathbf{5.67}); \\ b \sqsubseteq (\mathit{false}, \mathit{true}] \ \cap_L \ b \sqsubseteq [\mathit{false}, \mathit{true}] &= b \sqsubseteq (\mathit{false}, \mathit{true}]; \\ s \sqsubseteq [\{\mathbf{1}\}, \{\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}\}] \ \cap_L \ s \sqsubseteq [\{\mathbf{3}\}, \{\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{5}\}] &= s \sqsubseteq [\{\mathbf{1}, \mathbf{3}\}, \{\mathbf{1}, \mathbf{2}, \mathbf{3}\}]. \end{aligned}$$

4.4.2 Constraint Narrowing

Constraint narrowing is called constraint stabilisation and is mainly based on the intersection of simple interval constraints.

DEFINITION 4.30 *(Constraint store) A* constraint store S for X *is a finite subset of* $\mathcal{C}^X$*. The store* S *is* stable *if, for each* $x \in X$*, there is exactly one simple constraint* c *for* x *in* S*. The store* S *is* simple *if it contains only simple constraints. The store* S *is* inconsistent *if it contains at least one inconsistent (and thus simple) interval constraint, otherwise it is* consistent*.*

The domain of the simple constraint stores for X *is the set of all simple constraint stores for* X *and is denoted by* $\mathcal{S}^X$*. The* domain of the simple stable constraint stores for X *is the set of all simple stable constraint stores for* X *and is denoted by* $\mathcal{SS}^X$*.*

DEFINITION 4.31 *(Stabilised store) Let* $S \in \mathcal{S}^X$ *and, for each* $x \in X$*,* S_x *the set of constraints in* S *with constrained variable* x*. Then, the* stabilised store $S' \in \mathcal{SS}^X$ *of* S *is defined:*

$$S' = \{\cap_L(S_x) \mid x \in X \wedge x \in V_L \ \textit{(for some } L \in \mathcal{L}) \ \}$$

We write $S \mapsto S'$ *to express that* S' *is the stabilised store of* S*.*

Note that, by Definition 4.26, if $S_x = \emptyset$ then $\cap_L(S_x) = x \sqsubseteq \top_{R^b_L}$. This ensures that $S' \in \mathcal{SS}^X$.

EXAMPLE 4.32 *Suppose that* $Integer, \Re \in \mathcal{L}$ *and* $X = \{x, y, i\}$ *where* $x, y \in V_{\Re}$ *and* $i \in V_{Integer}$*. Let*

$$\begin{aligned} S = \{\ & x \sqsubseteq (\mathbf{8.3}, \mathbf{20.4}], \quad y \sqsubseteq [\mathbf{1.2}, \mathbf{10.5}), \quad i \sqsubseteq [\mathbf{0}, \mathbf{10}], \\ & x \sqsubseteq [\mathbf{1.0}, \mathbf{15.0}], \quad y \sqsubseteq (\mathbf{5.6}, \mathbf{15.3}), \quad i \sqsubseteq [\mathbf{2}, \mathbf{15}) \ \}, \\ S' = \{\ & x \sqsubseteq (\mathbf{8.3}, \mathbf{15.0}], \quad y \sqsubseteq (\mathbf{5.6}, \mathbf{10.5}), \quad i \sqsubseteq [\mathbf{2}, \mathbf{10}] \ \}. \end{aligned}$$

Then $S \mapsto S'$*.*

DEFINITION 4.33 *(A partial ordering on simple stable constraint stores) Let $S, S' \in \mathcal{SS}^X$ where c_x, c'_x denote the (simple) constraints for $x \in X$ in S and S', respectively. Then $S \preceq_s S'$ if and only if, for each $x \in X$, $c_x \preceq_{\mathcal{C}^X} c'_x$.*

Thus, $\mathcal{SS}^X$ forms a lattice with ordering $\preceq_s$, top element the set of type constraints for X (i.e., the set $\{x \sqsubseteq \top_{R^s_L} \mid x \in X \cap V_L, L \in \mathcal{L}\}$) and bottom element the set $\{x \sqsubseteq \bot_{R^s_L} \mid x \in X \cap V_L, L \in \mathcal{L}\}$.

EXAMPLE 4.34 *Consider the domains Integer and Bool. Then*

$$\big\{b \sqsubseteq [\mathit{false},\mathit{false}], i \sqsubseteq (\mathbf{2},\mathbf{4}]\big\} \prec_s \big\{b \sqsubseteq [\mathit{false},\mathit{true}], i \sqsubseteq [\mathbf{1},\mathbf{5})\big\}.$$

PROPOSITION 4.35 *Let $X' \subseteq X$ and $C \in \mathcal{S}^{X'}$. Let also $S, S' \in \mathcal{SS}^X$ such that $S \cup C \mapsto S'$. Then, $S' \preceq_s S$ and $\forall c \in C : \exists c' \in S' \,.\, c' \preceq_{\mathcal{C}^X} c$.*

PROOF 4.36 *Suppose for each $x \in X$, x is constrained by the constraints $c_x \in S$ and $c'_x \in S'$. Also, if $x \in X'$, suppose that C_x is the set of constraints in C with constrained variable x. Then, by Definition 4.31 on the preceding page,*

$$\begin{aligned} c'_x &= \cap_L(C_x \cup \{c_x\}) && \text{if } x \in X', \\ c'_x &= c_x && \text{otherwise.} \end{aligned}$$

Therefore, by Definition 4.28 on page 98 and the resulting contractance property, $c'_x \preceq_L c_x$ for each $x \in X$ (and thus by Definition 4.33, $S' \preceq_s S$) and, if $x \in X'$, $c'_x \preceq_{\mathcal{C}^X} c$ for all $c \in C_x$.

□

Any (in)consistent store in $\mathcal{SS}^X$ identifies a set of (in)consistent stores in $\mathcal{SS}^X$.

PROPOSITION 4.37 *Suppose $S, S' \in \mathcal{SS}^X$ where $S \preceq_s S'$. Then, if S' is inconsistent, S is also inconsistent.*

PROOF 4.38 *Suppose that S' is inconsistent. Then, by Definition 4.30 on the previous page, there is, at least, one inconsistent constraint $c'_x = x \sqsubseteq r' \in S'$ (for some $x \in X$). By Definition 4.26 on page 96, this means that r' is inconsistent.*

Let $c_x = x \sqsubseteq r$ be the constraint for x in S. By hypothesis, $S \preceq_s S'$ so that by Definition 4.33, $c_x \preceq_{\mathcal{C}^X} c'_x$ for all $x \in X$, and by Definition 4.26 on page 98, $r \preceq_{R^s_L} r'$. Thus, by Proposition 4.23, r is also inconsistent. Thus, by Definition 4.26, c_x is also inconsistent and hence, by Definition 4.30, S is inconsistent.

□

4.4.3 Constraint Propagation

Constraint propagation is defined by means of an evaluation function.

DEFINITION 4.39 *(Evaluating Indexical Terms) Let $S \in \mathcal{SS}^X$, $x \in X$, and let*

$$\mathcal{L}^X = \cup\{L^X \mid L \in \mathcal{L}, L^X = \{t \in L^b \mid vars(t) \subseteq X\}\},$$
$$\overline{\mathcal{L}^X} = \cup\{\overline{L^X} \mid L \in \mathcal{L}, \overline{L^X} = \{t \in \overline{L^b} \mid vars(t) \subseteq X\}\}$$

where $vars(t)$ denotes the set of variables occurring in t. Then the (overloaded) evaluation functions are defined:

$$eval :: \mathcal{SS}^X \times \mathcal{L}^X \to \mathcal{L}^X, \qquad eval :: \mathcal{SS}^X \times \overline{\mathcal{L}^X} \to \overline{\mathcal{L}^X},$$

$$\begin{aligned}
eval(S, t) &= t && \text{if } t \in L^s \cup \overline{L^s},\ L \in \mathcal{L},\\
eval(S, max(x)) &= t && \text{where } x \sqsubseteq \langle \overline{s}, t\rangle \in S,\\
eval(S, min(x)) &= \overline{s} && \text{where } x \sqsubseteq \langle \overline{s}, t\rangle \in S,\\
eval(S, val(x)) &= t && \text{if } x \sqsubseteq \langle \overline{t}, t\rangle \in S,\\
eval(S, val(x)) &= val(x) && \text{if } x \sqsubseteq \langle \overline{t}, t\rangle \notin S,\\
eval(S, \overline{val(x)}) &= \overline{t} && \text{if } x \sqsubseteq \langle \overline{t}, t\rangle \in S,\\
eval(S, \overline{val(x)}) &= \overline{val(x)} && \text{if } x \sqsubseteq \langle \overline{t}, t\rangle \notin S,\\
eval(S, \circ(t_1, \ldots, t_n)) &= \circ(eval(S, t_1), \ldots, eval(S, t_n)),\\
eval(S, \overline{\circ}(t_1, \ldots, t_n)) &= \overline{\circ}(eval(S, t_1), \ldots, eval(S, t_n)).
\end{aligned}$$

EXAMPLE 4.40 *Let*

$$S = \{x \sqsubseteq (\mathbf{1.23}, \mathbf{6.78}], y \sqsubseteq [\mathbf{1.54}, \mathbf{3.41}), i \sqsubseteq (\mathbf{1}, \mathbf{4}], b \sqsubseteq [false,false]\ \}$$

be a simple stable constraint store for $\{x, y, i, b\}$ i.e., $S \in \mathcal{SS}^{\{x,y,i,b\}}$. Then

$$\begin{aligned}
eval(S, \mathbf{2.34}]) &= \mathbf{2.34}],\\
eval(S, min(i)\overline{+}[\mathbf{3}) &= eval(S, min(i))\overline{+}eval(S, [\mathbf{3}) = (\mathbf{1}\overline{+}[\mathbf{3} = (\mathbf{4},\\
eval(S, trunc(max(x))) &= trunc(eval(S, max(x))) = trunc(\mathbf{6.78}]) = \mathbf{6}],\\
eval(S, \overline{val(b)}) &= [false,\\
eval(S, val(x)) &= val(x).
\end{aligned}$$

where $+/2$ (for $L = Integer$) and $trunc/1$ are as defined in Examples 4.11 and 4.14.

Note that our indexical terms are a generalisation of the indexical terms provided by clp(FD) (Codognet and Diaz, 1996a) and allow for infinite as well as finite ranges.

Let $c = x \sqsubseteq \langle \overline{s}, t\rangle$ be an interval constraint in $\mathcal{C}_L^X$ and thus $\overline{s} \in \overline{L^b}$ and $t \in L^b$ for some $L \in \mathcal{L}$. Then we overload $eval/2$ and define $eval(S, c)$ as

$$eval(S, c) = x \sqsubseteq \big\langle eval(S, \overline{s}), eval(S, t)\big\rangle.$$

DEFINITION 4.41 *(Constraint propagation) Suppose* $S \in \mathcal{SS}^X$.

If $c, c' \in \mathcal{C}_L^X$ *where* c' *is simple and* $eval(S, c) = c'$, *then we say that* c is propagated (using S) *to* c' *and write* $c \rightsquigarrow^S c'$.

If $C \subseteq \mathcal{C}^X$ *and* $C' = \{c' \mid \exists c \in C \,.\, c \rightsquigarrow^S c'\}$ *then we say that* C is propagated to C' using S *and write* $C \rightsquigarrow^S C'$. *As a consequence* $C' \subseteq \mathcal{C}^{X'}$ *is a simple constraint store where* $X' \subseteq X$.

Note that, if $x \sqsubseteq \langle \overline{s}, t \rangle \in S$ where $s \neq t$, then the evaluation function *eval* applied to $val(x)$ (resp. $\overline{val(x)}$) returns $val(x)$ (resp. $\overline{val(x)}$) unchanged. Thus the indexical $val(x)$ (resp. $\overline{val(x)}$) provides a useful mechanism to delay the propagation of constraints. (Subsection 5.3.1 gives an example of their use.)

Observe also that if C is finite and $C \rightsquigarrow^S C'$, then C' is computable (i.e., the process of propagating C to C' using S terminates).

EXAMPLE 4.42 *Consider both the store* S' *and the variables in Example 4.32 on page 99, and the operator* $trunc/1$ *as defined in Example 4.14 on page 90. Then*

$$\Big\{x \sqsubseteq min(y)\text{,}\mathbf{20.4}],\ i \sqsubseteq \overline{trunc}\big(min(y)\big)\text{,}trunc\big(max(y)\big)\Big\} \rightsquigarrow^{S'} \big\{x \sqsubseteq (\mathbf{5.6}, \mathbf{20.4}],\ i \sqsubseteq [\mathbf{5}, \mathbf{10}]\big\}.$$

4.4.4 Equivalence in the Discrete Domain

When the computation domain L is discrete, we can identify equivalent elements in the bounded computation domain L^s and its mirror $\overline{L^s}$ and hence, the interval domain R_L^s and the constraint domain $\mathcal{C}_L^X$.

EXAMPLE 4.43 *For any* $i \in Integer$, *the immediate predecessor is* $i - 1$ *so that, for instance,* $\mathbf{3}) = \mathbf{2}]$. *Similarly, with the Bool domain, the immediate predecessor of true is false so that* $\mathbf{true}) = \mathbf{false}]$.

Suppose that L is a discrete domain in which the *immediate predecessor* $pre(\mathrm{a})$ of every value $\mathrm{a} \in L$, with $\mathrm{a} \neq \bot_L$, is defined. To reduce the size of the domain L^s, we introduce the following *equivalence rule* for any $\mathbf{a}) \in L^s$ and $\mathrm{a} \neq \bot_L$

$$\mathbf{a}) \equiv pre(\mathrm{a})] \quad \text{in } L^s.$$

By the duality principle of lattices, in the domain $\overline{L^s}$ (where $succ(\mathrm{a})$ is the *immediate successor* of any $\mathrm{a} \in L$ with $\mathrm{a} \neq \top_L$) we have

$$(\mathbf{a} \equiv [succ(\mathrm{a}) \quad \text{in } \overline{L^s}.$$

If $\bot_L$ and $\top_L$ elements were added as fictitious bounds, then we define

$$pre(\top_L) \equiv \top_L, \qquad succ(\bot_L) \equiv \bot_L.$$

When the interval domain is constructed from a discrete domain L, the equivalence rules allows a reduction in the size of L^s and $\overline{L^s}$ and thus the size of the interval domain R_L^s. If $pre/1$ is defined for every element of L, then they provide a canonical form for L^s where '(',')' brackets are eliminated in favour of the '[',']'.

EXAMPLE 4.44 *For the Integer domain,* $[\mathbf{1},\mathbf{3}) \equiv [\mathbf{1},\mathbf{2}]$. *Similarly, for the Bool domain,* $[false, true) \equiv [false, false]$. *Suppose* $L = \{0,1,2,3\}$ *is a lattice where* $0 < 1 < 2 < 3$. *Then,*

$$L^s = \{\ \mathbf{0}),\mathbf{0}],\mathbf{1}),\mathbf{1}],\mathbf{2}),\mathbf{2}],\mathbf{3}),\mathbf{3}]\ \} \equiv \{\ \mathbf{0}),\mathbf{0}],\mathbf{1}],\mathbf{2}],\mathbf{3}]\ \},$$
$$\overline{L^s} = \{\ (\mathbf{3},[\mathbf{3},(\mathbf{2},[\mathbf{2},(\mathbf{1},[\mathbf{1},(\mathbf{0},[\mathbf{0}\ \} \equiv \{\ (\mathbf{3},[\mathbf{3},[\mathbf{2},[\mathbf{1},[\mathbf{0}\ \}.$$

Suppose also that $X = \{x, b\}$, $x \in V_{Integer}$ *and* $b \in V_{Bool}$. *Then,* $x \sqsubseteq (\mathbf{1},\mathbf{5})$ *is equivalent to* $x \sqsubseteq [\mathbf{2},\mathbf{4}]$ *in* $\mathcal{C}^X_{Integer}$ *and* $b \sqsubseteq (false, true]$ *is equivalent to* $b \sqsubseteq [true, true]$ *in* $\mathcal{C}^X_{Bool}$.

With these rules for discrete domains more inconsistencies can be detected.

EXAMPLE 4.45 *Consider again the domain L in Example 4.25. Suppose L is discrete and that* $pre(c) = \mathbf{a}$ *and* $pre(b) = \mathbf{c}$. *Then the ranges* $(\mathbf{a},\mathbf{c}) \in R^s_L$ *and* $(\mathbf{c},\mathbf{b})$ *are inconsistent since they are equivalent to ranges* $[\mathbf{c},\mathbf{a}]$ *and* $[\mathbf{b},\mathbf{c}]$ *respectively. These are the circled nodes in Figure 4.2.*

For instance, let $L = Integer$. *Then the range* $(\mathbf{1},\mathbf{2})$ *is inconsistent in the domain* $R^s_{Integer}$ *since* $[\mathbf{2},\mathbf{1}]$ *is inconsistent. Also the constraint* $x \sqsubseteq (\mathbf{1},\mathbf{2})$ *is inconsistent since* $x \sqsubseteq [\mathbf{2},\mathbf{1}]$ *is inconsistent.*

In the rest of the document, we assume that all the simple interval constraints defined on discrete domains are reduced (wherever possible) by the equivalence shown in this section.

4.4.5 A Solution for a Constraint Store

DEFINITION 4.46 *(Solution) Let* $C \in \wp(\mathcal{C}^X)$ *be a constraint store for X. A* solution for C *is a consistent store* $R \in \mathcal{SS}^X$ *where,*

$$C \rightsquigarrow^R C'$$
$$R \cup C' \mapsto R.$$

The set of all solutions for C is denoted as $Sol(C)$. *If it exists, the* most general solution for C *(in short* m.g.s. of C*), is defined as:*

$$m.g.s.\ of\ C = G \in Sol(C) \mid \forall R \in Sol(C).R \preceq_s G.$$

Therefore, a solution is a consistent constraint store containing, for each of the constrained variables, exactly one simple interval constraint that cannot be reduced by means of the propagation or stabilisation procedures.

LEMMA 4.47 *Let* $C \in \wp(\mathcal{C}^X)$ *and* $S, R \in \mathcal{SS}^X$. *If R is a solution for* $C \cup S$, *then,* $R \preceq_s S$.

PROOF 4.48 *By Definition 4.46 on the preceding page, R is consistent, and*

$$C \cup S \rightsquigarrow^R C_R \text{ and } R \cup C_R \mapsto R. \tag{4.8}$$

By Definition 4.41 on page 102, if $C \cup S \rightsquigarrow^R C_R$, then C_R is equivalent to $C_R = C_1 \cup C_2$ where

$$C \rightsquigarrow^R C_1 \text{ and } S \rightsquigarrow^R C_2,$$

and also

$$C_2 = \{c' \mid \exists c \in S \,.\, c \rightsquigarrow^R c'\}.$$

Observe that if $c = x \sqsubseteq \langle \overline{s}, t\rangle$ is a simple constraint and $x \in V_L$ then, by Definition 4.26 on page 96, $\langle \overline{s}, t\rangle \in R^s_L$ and, by Definition 4.19 on page 92, $\overline{s} \in \overline{L^s}$ and $t \in L^s$. Therefore

$$eval(R, c) =^1 x \sqsubseteq \big\langle eval(R, \overline{s}), eval(R, t)\big\rangle =^2 x \sqsubseteq \langle \overline{s}, t\rangle \;=\; c.$$

where $=^1$ comes from (overloaded) definition of eval function for constraints in Page 101 and $=^2$ comes from Definition 4.39 on page 101. By Definition 4.41 on page 102, this means that $c \rightsquigarrow^R c$ if c is simple. Since S contains only simple constraints, then by Definition 4.41, $C_2 = S$.

Moreover, from (4.8)*, $R \cup C_R \mapsto R$ and, as shown previously, $C_R = C_1 \cup C_2 = C_1 \cup S$. Thus $R \cup C_1 \cup S \mapsto R$ and, by Proposition 4.35 on page 100, $R \preceq_s S$.*

□

4.4.6 Monotonicity of Constraints

In this Section we show that all the constraints in our framework are intrinsically monotonic.

LEMMA 4.49 *Suppose that $c \preceq_{\mathcal{C}^X} c'$ are simple consistent constraints for $L \in \mathcal{L}$ constraining the same variable $y \in X$ and suppose also that $c' = y \sqsubseteq \langle \overline{t'}, t'\rangle$ for some $t' \in L^s$. Then $c = c'$.*

PROOF 4.50 *Suppose that $t' = \mathbf{a}\}$, for some $\mathbf{a} \in L$. Then, as c' is consistent, by Definition 4.21 on page 93, $t' \neq \mathbf{a})$ so that $t' = \mathbf{a}]$. Suppose $c = y \sqsubseteq \langle \overline{s}, t\rangle$. Then, by Definition 4.26 on page 98, $\langle \overline{s}, t\rangle \preceq_{R^s_L} \langle \overline{\mathbf{a}]}, \mathbf{a}]\rangle$ and, by Definition 4.19 on page 92,*

$$\overline{s} \preceq_{\overline{L^s}} \overline{\mathbf{a}]} \text{ and } t \preceq_{L^s} \mathbf{a}].$$

Then, by Definition 4.4 on page 86 and by the product of lattices (i.e., the lexicographic product),

$$\begin{aligned} \overline{s} = \overline{\mathbf{a}]} \text{ or } \overline{s} = \overline{\mathbf{a})} \text{ or } \overline{s} = \overline{\mathbf{a_1}\}} \text{ and } \hat{\mathbf{a}}_1 \prec_{\hat{L}} \hat{\mathbf{a}}, \\ t = \mathbf{a}] \text{ or } t = \mathbf{a}) \text{ or } t = \mathbf{a_2}\} \text{ and } \mathbf{a}_2 \prec_L \mathbf{a}. \end{aligned}$$

By the duality principle of lattices in Section 4.2, this is equivalent to

$$s = \mathbf{a}] \text{ or } s = \mathbf{a}) \text{ or } s = \mathbf{a_1}\} \text{ and } \mathbf{a} \prec_L \mathbf{a}_1,$$
$$t = \mathbf{a}] \text{ or } t = \mathbf{a}) \text{ or } t = \mathbf{a_2}\} \text{ and } \mathbf{a}_2 \prec_L \mathbf{a}.$$

However, c is consistent so that, by Definition 4.26 on page 96, $\langle \overline{s}, t\rangle$ is consistent. By Definition 4.21, this means that $s \preceq_{L^s} t$ and, if $s = \mathbf{a})$ then $t \neq \mathbf{a}]$. The only case, for which this holds is when $s = t = \mathbf{a}]$.

□

Each interval constraint is propagated monotonically with respect to the ordering on simple stable stores.

LEMMA 4.51 *Let $S_1, S_2 \in \mathcal{SS}^X$ be two consistent stores such that $S_1 \preceq_s S_2$ and $c, c_2 \in \mathcal{C}^X$ such that $c \rightsquigarrow^{S_2} c_2$. Then, there exists $c_1 \in \mathcal{C}^X$ such that $c \rightsquigarrow^{S_1} c_1$ and $c_1 \preceq_{\mathcal{C}^X} c_2$.*

PROOF 4.52 *Let $c = x \sqsubseteq \langle \overline{s}, t\rangle$ where $x \in X$ and $x \in V_L$ for some $L \in \mathcal{L}$. Then as $c \rightsquigarrow^{S_2} c_2$, by Definition 4.41 on page 102, $c_2 = eval(S_2, c)$ and c_2 is simple. Then it follows from the (overloaded) definition of eval/2 for constraints shown in Page 101 and Definition 4.26 on page 96 that*

$$c_2 = x \sqsubseteq \big\langle eval(S_2, \overline{s}), eval(S_2, t)\big\rangle,$$
$$eval(S_2, \overline{s}) \in \overline{L^s} \quad \text{and} \quad eval(S_2, t) \in L^s. \tag{4.9}$$

Suppose that $c_1 = x \sqsubseteq eval(S_1, c)$. Then, again it follows from the (overloaded) definition of eval/2 for constraints that

$$c_1 = x \sqsubseteq \big\langle eval(S_1, \overline{s}), eval(S_1, t)\big\rangle.$$

We have to prove that $c \rightsquigarrow^{S_1} c_1$ and $c_1 \preceq_{\mathcal{C}^X} c_2$ which means that, by Definitions 4.26 and 4.41, we have to show that c_1 is simple and that

$$\big\langle eval(S_1, \overline{s}), eval(S_1, t)\big\rangle \preceq_{R_L^s} \big\langle eval(S_2, \overline{s}), eval(S_2, t)\big\rangle. \tag{4.10}$$

However, by Definition 4.19 on page 92, if (4.10) holds, c_1 is simple. Thus, by the product of lattices (i.e., direct product), we just have to show that

$$eval(S_1, \overline{s}) \preceq_{\overline{L^s}} eval(S_2, \overline{s}) \text{ and} \tag{i}$$
$$eval(S_1, t) \preceq_{L^s} eval(S_2, t). \tag{ii}$$

Let $n(v)$ be the number of constraint operators in v. We prove (i) by induction on $n(\overline{s})$. The proof of (ii) is similar and omitted.

- Base case: $n(\overline{s}) = 0$.

  If $\overline{s} \in \overline{L^s}$, *then, by Definition 4.39 on page 101,* $eval(S_1, \overline{s}) = eval(S_2, \overline{s}) = \overline{s}$.

  If $\overline{s} \notin \overline{L^s}$, *then* $\overline{s} = min(y)$ *or* $\overline{s} = \overline{val(y)}$ *for some* $y \in X$. *Observe that there exists* $c_y = y \sqsubseteq \langle \overline{s_y}, t_y \rangle \in S_1$ *and* $c'_y = y \sqsubseteq \langle \overline{s'_y}, t'_y \rangle \in S_2$ *so that as, by hypothesis,* $S_1 \preceq_s S_2$ *we have*

$$S_1 \preceq_s S_2 \Rightarrow^1 c_y \preceq_{\mathcal{C}^X} c'_y \Rightarrow^2 \langle \overline{s_y}, t_y \rangle \preceq_{R_L^s} \langle \overline{s'_y}, t'_y \rangle \Rightarrow^3 \overline{s_y} \preceq_{\overline{L^s}} \overline{s'_y} \text{ and } t_y \preceq_{L^s} t'_y. \tag{4.11}$$

  where $\Rightarrow^1$ *follows from Definition 4.33 on page 100,* $\Rightarrow^2$ *follows from Definition 4.26, and* $\Rightarrow^3$ *follows from the product of lattices (i.e., direct product) and Definition 4.19.*

  Suppose first that $\overline{s} = min(y)$. *Then, by Definition 4.39,*

$$eval(S_1, \overline{s}) = \overline{s_y} \text{ and } eval(S_2, \overline{s}) = \overline{s'_y}.$$

  Therefore, by (4.11), $eval(S_1, \overline{s}) \preceq_{\overline{L^s}} eval(S_2, \overline{s})$.

  Now suppose that $\overline{s} = \overline{val(y)}$. *By* (4.9) $eval(S_2, \overline{s}) \in \overline{L^s}$ *and by Definition 4.39,* $eval(S_2, \overline{s}) = \overline{s'_y}$ *and* $s'_y = t'_y$. *Therefore, as* S_1 *is consistent by Definition 4.30* c_y *is also consistent, it follows from* (4.11) *and Lemma 4.49, that* $s_y = t_y = s'_y$. *Thus, by Definition 4.39,* $eval(S_1, \overline{s}) = \overline{s_y}$ *so that* $eval(S_1, \overline{s}) = eval(S_2, \overline{s})$.

- Non-base case: $n(\overline{s}) > 0$.

  Suppose $\circ :: L_1^s \times \ldots \times L_n^s \to L^s$ *is a constraint operator (for* L^s*). Then, by Proposition 4.12*(b) *in Page 89,* $\overline{\circ}$ *is also a constraint operator. Then,*

$$\begin{aligned} n(\overline{s}) > 0 \Rightarrow \overline{s} = & \overline{\circ(s_1, \ldots, s_n)} \;=^1\; \overline{\circ}(\overline{s_1}, \ldots, \overline{s_n}) \\ & \Rightarrow^4 \begin{cases} eval(S_1, \overline{s}) = \overline{\circ}(eval(S_1, \overline{s_1}), \ldots, eval(S_1, \overline{s_n})); \\ eval(S_2, \overline{s}) = \overline{\circ}(eval(S_2, \overline{s_1}), \ldots, eval(S_2, \overline{s_n})); \end{cases} \end{aligned}$$

  where $=^1$ *comes from Definition 4.10 and* $\Rightarrow^4$*comes from Definition 4.39 on page 101. By the inductive hypothesis,*

$$eval(S_1, \overline{s_i}) \preceq_{\overline{L_i^s}} eval(S_2, \overline{s_i}),\ i \in \{1, \ldots, n\},$$

  and by Proposition 4.12(a) *in Page 89,* $\overline{\circ}$ *is monotonic so that* (i) *holds.*

□

PROPOSITION 4.53 *Let* $S_1, S_2 \in \mathcal{SS}^X$ *such that* S_1 *and* S_2 *are consistent and* $S_1 \preceq_s S_2$ *and* $C \in \wp(\mathcal{C}^X)$ *such that*

$$C \rightsquigarrow^{S_1} C_1 \text{ and } S_1 \cup C_1 \mapsto S'_1,$$
$$C \rightsquigarrow^{S_2} C_2 \text{ and } S_2 \cup C_2 \mapsto S'_2.$$

Then $S'_1 \preceq_s S'_2$.

PROOF 4.54 *By hypothesis* $C \rightsquigarrow^{S_1} C_1$ *and* $C \rightsquigarrow^{S_2} C_2$ *so that, by Definition 4.41 on page 102,* $C_1 \in \mathcal{C}^{X_1}$ *and* $C_2 \in \mathcal{C}^{X_2}$ *where* $X_1 \subseteq X$ *and* $X_2 \subseteq X$ *and*

$$C_1 = \{c_1 \mid \exists c \in C \,.\, c \rightsquigarrow^{S_1} c_1\} \quad \text{and} \quad C_2 = \{c_2 \mid \exists c \in C \,.\, c \rightsquigarrow^{S_2} c_2\}.$$

As $S_1 \preceq_s S_2$*, by Lemma 4.51 on page 105,*

$$\forall c_2 \in C_2 : \exists c_1 \in C_1 \text{ such that } c_1 \preceq_{\mathcal{C}^X} c_2. \tag{4.12}$$

By Definition 4.26 on page 98, if $c_1 \preceq_{\mathcal{C}^X} c_2$ *then* c_1 *and* c_2 *are constrained on the same variable* $x \in X$*. Then, it follows from* (4.12) *that*

$$X_2 \subseteq X_1. \tag{4.13}$$

Let C_{1x} *and* C_{2x} *be the sets of constraints, in* C_1 *and* C_2 *respectively, with constrained variable* $x \in X$ *(note that* C_{1x} *and* C_{2x} *can be the empty set). It follows from* (4.12) *and* (4.13) *that, for each* $c_2 \in C_{2x}$*, there exists* $c_1 \in C_{1x}$ *such that* $c_1 \preceq_{\mathcal{C}^X} c_2$*.*

Suppose that c_{1x}, c_{2x}, c'_{1x} *and* c'_{2x} *are the constraints for* $x \in X$ *in the stores* S_1, S_2, S'_1 *and* S'_2 *respectively. By hypothesis* $S_1 \preceq_s S_2$ *so that, by Definition 4.33,* $c_{1x} \preceq_{\mathcal{C}^X} c_{2x}$*. Since* $S_1 \cup C_1 \mapsto S'_1$ *and* $S_2 \cup C_2 \mapsto S'_2$*, by Definition 4.31 on page 99,*

$$c'_{1x} = \cap_L(C_{1x} \cup \{c_{1x}\}),$$
$$c'_{2x} = \cap_L(C_{2x} \cup \{c_{2x}\}).$$

As consequence of Definition 4.28 on page 98 and contractance property of $\cap_L$*,*

$$c'_{1x} \preceq_{\mathcal{C}^X} c'_{2x}, \text{ for each } x \in X.$$

Therefore, by Definition 4.33, $S'_1 \preceq_s S'_2$*.* □

EXAMPLE 4.55 *Consider the definition of the operator* $-$ *in Example 4.11 on page 88, when* L *is* $\Re$ *so that* $- :: \Re^s \times \overline{\Re^s} \to \Re^s$*. Suppose that* $X = \{x, y\}$*,* $x, y \in V_{\Re}$*,*

$$S_1 = \{\ y \sqsubseteq (\mathbf{2.0}, \mathbf{4.0}],\ x ::' \Re\ \} \quad \text{and} \quad S_2 = \{\ y \sqsubseteq (\mathbf{1.0}, \mathbf{11.0}],\ x ::' \Re\ \}.$$

Then $S_1, S_2 \in \mathcal{SS}^X$ *and* $S_1 \prec_s S_2$*. Suppose that*

$$c_1 = x \sqsubseteq [\mathbf{0.0}, \mathbf{20.0}]\text{–}min(y) \quad \text{and} \quad c_2 = x \sqsubseteq [\mathbf{0.0}, \mathbf{20.0}]\text{–}max(y).$$

Then c_1 *is an interval constraint for* $\Re$ *because*

$$[\mathbf{0.0} \in \overline{\Re^b} \text{ and } \mathbf{20.0}]\text{–}min(y) \in \Re^b$$

and hence $[\mathbf{0.0}, \mathbf{20.0}]\text{–}min(y) \in R^b_{\Re}$*. Using the constraint propagation procedure for* S_1 *and* S_2 *we obtain*

$$c_1 \rightsquigarrow^{S_1} c_{11}, \qquad c_1 \rightsquigarrow^{S_2} c_{12}$$

where

$$c_{11} = x \sqsubseteq [\mathbf{0.0}, \mathbf{18.0}), \qquad c_{12} = x \sqsubseteq [\mathbf{0.0}, \mathbf{19.0}).$$

Then we have $c_{11} \prec_{\mathcal{C}^X} c_{12}$ *(i.e.,* c_1 *was propagated monotonically with respect to the ordering of constraint stores).*

However, c_2 *is not an interval constraint. This is because, although* $[\mathbf{0.0} \in \overline{\Re^b}$ *and* $\mathbf{20.0}] \in \Re^b$*, we have* $max(y) \notin \overline{\Re^b}$ *with the consequence that,*

$$\mathbf{20.0}]\text{–}max(y) \notin \Re^b$$

and hence

$$[\mathbf{0.0}, \mathbf{20.0}]\text{–}max(y) \notin R^b_{\Re}.$$

Observe that applying the constraint propagation procedure we would obtain

$$c_2 \rightsquigarrow^{S_1} c_{21}, \qquad c_2 \rightsquigarrow^{S_2} c_{22}$$

where

$$c_{21} = x \sqsubseteq [\mathbf{0.0}, \mathbf{16.0}], \qquad c_{22} = x \sqsubseteq [\mathbf{0.0}, \mathbf{9.0}].$$

Then we have $c_{22} \prec_{\mathcal{C}^X} c_{21}$*. Thus the constraint procedure applied to* c_2 *using the smallest constraint store* S_1 *derives the largest range for* x*. The problem is caused by the fact that if* S_2 *is replaced by a smaller store such as* S_1*,* $max(y)$ *also decreases in* $\Re^s$*, so that the value of* $\mathbf{20.0}]\text{–}max(y)$ *actually increases. Thus, the right bound of the range for* x *in* c_2 *also increases so that the upper limit for* y *could never be reduced.*

Fortunately this problem is detected by a simple check of the validity of the interval constraints in our theoretical framework. Observe that the acceptability of expressions such as c_2 *as valid constraints can be decided* a priori *using standard type-checking techniques.*

4.5 Operational Semantics

In this section, we provide an operational schema for propagating the interval constraints and prove both its correctness and termination.

We continue to use L to denote any domain in $\mathcal{L}$, $X \in \wp(\mathcal{V}_{\mathcal{L}})$ the set of constrained variables, $\mathcal{C}^X$ the set of all interval constraints domain for X and $\mathcal{SS}^X$ the set of all simple stable constraint stores for X.

4.5.1 Operational Schema for Constraint Propagation

Let $C \in \wp_f(\mathcal{C}^X)$ and $S \in \mathcal{SS}^X$. We define here $solve(C, S)$, an *operational schema* for computing a solution (if it exists) for $C \cup S$. This schema is shown in Figure 4.3.

If at least one solution exists, the store S contains a solution for $C \cup S$ (assuming termination of the schema).

procedure $solve(C, S)$
begin
  if S *is consistent then* (0)
    $C := C \cup S$; (1)
    repeat
      $C \rightsquigarrow^S C'$; %% Constraint Propagation (2)
      $S' := S$; (3)
      $S' \cup C' \mapsto S$; %% Store stabilisation (4)
    until S *is inconsistent or* $S = S'$; (5)
  endif;
endbegin.

Figure 4.3: *solve*/2: a generic schema for interval constraint propagation

THEOREM 4.56 *(Correctness) Let* $C \in \wp_f(\mathcal{C}^X)$ *and* $S \in \mathcal{SS}^X$. *If at least one solution for* $C \cup S$ *exists, a terminating execution of the operational schema for* $solve(C, S)$ *computes in* S *the most general solution for* $C \cup S$. *Otherwise, if there is no solution for* $C \cup S$, $solve(C, S)$ *computes in* S *an inconsistent store.*

PROOF 4.57 *Let* S_0 *be the initial value of* S *and, by step* (1), $C = C \cup S_0$. *Suppose that there are* k *iterations of the repeat loop and that, for each* i *where* $1 \leq i \leq k$, S_i *is the value of the constraint store* S *at step (5), after completing* i *iterations of the repeat loop.*

Suppose first that a solution R *(for* $C \cup S_0$*) exists. Then, by Definition 4.46,* R *is consistent,* $R \in \mathcal{SS}^X$ *and*

$$C \rightsquigarrow^R C_R \text{ and } R \cup C_R \mapsto R. \tag{4.14}$$

Note that, initially $S_0 = S \in \mathcal{SS}^X$. *Then, by Lemma 4.47 on page 103,*

$$R \preceq_s S_0. \tag{4.15}$$

We show by induction on i, *that after* $i \geq 0$ *iterations of the* repeat *loop*

$$R \preceq S_i. \tag{4.16}$$

The base case when $i = 0$ *is given by* (4.15). *For the inductive step, suppose that there are at least* $i > 0$ *iterations of the repeat loop and that, after* $i - 1$ *steps, we have* $R \preceq S_{i-1}$. *Then*

$$C \rightsquigarrow^{S_{i-1}} C', \qquad \textit{step } (2); \tag{4.17}$$
$$S_{i-1} \cup C' \mapsto S_i, \qquad \textit{step } (4). \tag{4.18}$$

It follows from (4.14), (4.17), (4.18)*, and Proposition 4.53 on page 106 that* (4.16) *holds.*

Therefore, $R \preceq S_k$. As R is consistent, by Proposition 4.37 on page 100, S_k is consistent. However, the procedure terminates before the $k+1$'th iteration so that the test in step (5) *is true and we must have $S_k = S_{k-1}$.*

By (4.17) *and* (4.18)

$$C \rightsquigarrow^{S_{k-1}} C' \text{ and } S_{k-1} \cup C' \mapsto S_k. \tag{4.19}$$

Thus, by Definition 4.46, S_k is a solution for C (i.e., $C \cup S_0$). Moreover, if R is another solution for $C \cup S_0$, then as $R \preceq_s S_k$ and as R was any solution for $C \cup S_0$, S_k is the most general solution for $C \cup S_0$.

Suppose next that there is no solution for $C \cup S_0$. Then $S_{k-1} \neq S_k$ or else, by (4.19)*, S_k would be a solution. Thus, in this case, as the procedure terminates before the $k+1$'th iteration so that the test in step* (5) *is true, we must have S_k is inconsistent.*

□

We do not discuss possible efficiency improvements since the main aim is to provide the basic methodology, showing how the execution method of clp(FD) may be generalised for constraint propagation on any lattice-structure domain.

4.5.2 Termination

New constraints, created by the propagation step (2) (see Figure 4.3) are added to the set of constraints before the stabilisation step (4). Thus, with infinite domains, the algorithm may not terminate (note that the constraints could be contracted indefinitely in the stabilisation step).

EXAMPLE 4.58 *Consider the operator $div2_{\Re} :: \Re \to \Re$ where $div2_{\Re}\ \mathbf{a} = \frac{\mathbf{a}}{2.0}$ and let $div2_B$ be the identity on B. Then let C be the constraint store*

$$\begin{array}{l} \{\ x \sqsubseteq [\mathbf{0.0}, \mathbf{10.0}],\ x \sqsubseteq [\mathbf{0.0}, div2(max(y)), \\ \quad y \sqsubseteq [\mathbf{0.0}, \mathbf{10.0}],\ y \sqsubseteq [\mathbf{0.0}, div2(max(x))\ \} \end{array}$$

where $x, y \in V_{\Re}$ and S_0 the top element of the lattice $\mathcal{SS}^{\{x,y\}}$. Let also S_i be the value of the store S at the end of the i'th iteration for $i \geq 1$ of the operational schema for $solve(C, S)$ with S_0 the initial value of S. Then, in the execution of $solve(C, S)$, S is indefinitely reduced i.e.,

$$\begin{array}{l} S_0 = \{\ x \sqsubseteq [\bot_{\Re}, \top_{\Re}], y \sqsubseteq [\bot_{\Re}, \top_{\Re}]\ \}, \\ S_1 = \{\ x \sqsubseteq [\mathbf{0.0}, \mathbf{10.0}],\ y \sqsubseteq [\mathbf{0.0}, \mathbf{10.0}]\ \}, \\ S_2 = \{\ x \sqsubseteq [\mathbf{0.0}, \mathbf{5.0}],\ y \sqsubseteq [\mathbf{0.0}, \mathbf{5.0}]\ \}, \\ S_3 = \{\ x \sqsubseteq [\mathbf{0.0}, \mathbf{2.5}],\ y \sqsubseteq [\mathbf{0.0}, \mathbf{2.5}]\ \}, \\ \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \end{array}$$

To force termination, we introduce the notion of *precision of a consistent constraint.*

DEFINITION 4.59 *(Precision of a constraint) Let $\mathcal{CC}_L^X$ be the set of all consistent (and thus simple) interval constraints for L with constrained variables in X, $x \in X \cap V_L$ for any $L \in \mathcal{L}$ and $\Re\mathcal{I}$ denote the lexicographic product $(\Re^+, Integer)$ where $\Re^+$ is the domain of non-negative reals. Then we define*

$$\begin{array}{l} precision_L :: \mathcal{CC}_L^X \to \Re\mathcal{I} \\ precision_L(\ x \sqsubseteq \{_1 a, b\}_2\ ) = (\ \hat{a} \diamond_L b,\ \}_1 \diamond_B \}_2\ ) \end{array} \tag{4.20}$$

where $\diamond_L :: \{\, (\hat{a}, b) \mid a, b \in L, a \preceq_L b \,\} \to \Re^+$ is a (system or user defined) strict monotonic function and $\diamond_B :: B \times B \to \{0, 1, 2\}$ is the strict monotonic function

$$\text{`]'} \diamond_B \text{`]'} \stackrel{\text{def}}{=} 2 \qquad \text{`]'} \diamond_B \text{`)'} \stackrel{\text{def}}{=} 1 \qquad \text{`)'} \diamond_B \text{`]'} \stackrel{\text{def}}{=} 1 \qquad \text{`)'} \diamond_B \text{`)'} \stackrel{\text{def}}{=} 0.$$

Since $precision_L$ is defined only on consistent constraints, the function $\diamond_L$ only needs to be defined when its first argument is smaller or equal than the second. This function has to be defined for each computation domain.

PROPOSITION 4.60 *$precision_L$ is strict monotonic. i.e.,*

$$precision_L(c) <_{\Re\mathcal{I}} precision_L(c') \text{ if } c <_{\mathcal{CC}_L^X} c'.$$

PROOF 4.61 *Suppose that $c = x \sqsubseteq \{\mathbf{1}\, a, b\}_\mathbf{2}$ and $c' = x \sqsubseteq \{'_\mathbf{1}\, a', b'\}'_\mathbf{2}$ where $c \prec_{\mathcal{CC}_L^X} c'$. Then, we have to prove that*

$$precision_L(c) <_{\Re\mathcal{I}} precision_L(c'),$$

which, by Definition 4.59, is equivalent to showing:

$$(\ \hat{a} \diamond_L b, \}_1 \diamond_B \}_2\ ) <_{\Re\mathcal{I}} (\ \hat{a'} \diamond_L b', \}'_1 \diamond_B \}'_2\ ). \tag{4.21}$$

By hypothesis, $c \prec_{\mathcal{CC}_L^X} c'$, so that by Definition 4.26 on page 98

$$\{\mathbf{1}\, a, b\}_\mathbf{2} \prec_{R_L^s} \{'_\mathbf{1}\, a', b'\}'_\mathbf{2}$$

and, by the product of lattices (i.e., direct product), either

$$\{\mathbf{1}\, a \prec_{\overline{L^s}} \{'_\mathbf{1}\, a' \quad \text{and} \quad b\}_\mathbf{2} \preceq_{L^s} b'\}'_\mathbf{2} \text{ or} \tag{4.22}$$

$$\{\mathbf{1}\, a \preceq_{\overline{L^s}} \{'_\mathbf{1}\, a' \quad \text{and} \quad b\}_\mathbf{2} \prec_{L^s} b'\}'_\mathbf{2}. \tag{4.23}$$

If (4.22) holds, then, by the product of lattices (i.e., lexicographic product),

$$(\ \hat{a} \prec_{\hat{L}} \hat{a'} \text{ or } \hat{a} =_{\hat{L}} \hat{a'} \text{ and } \}_1 \prec_B \}'_1\ )\ \text{and}\ (\ b \prec_L b' \text{ or } b =_L b' \text{ and } \}_2 \preceq_B \}'_2\ ). \tag{4.24}$$

Similarly, if (4.23) holds, then,

$$(\ \hat{a} \prec_{\hat{L}} \hat{a'} \text{ or } \hat{a} =_{\hat{L}} \hat{a'} \text{ and } \}_1 \preceq_B \}'_1\ )\ \text{and}\ (\ b \prec_L b' \text{ or } b =_L b' \text{ and } \}_2 \prec_B \}'_2\ ). \tag{4.25}$$

Therefore, there are three cases: $\hat{a} \prec_{\hat{L}} \hat{a'}$ *and* $b \preceq_L b'$, $\hat{a} = \hat{a'}$ *and* $b \prec_L b'$, *or* $\hat{a} = \hat{a'}$ *and* $b = b'$. *However, we have*

$$\left\{ \begin{array}{l} \hat{a} \prec_{\hat{L}} \hat{a'} \text{ and } b \preceq_L b' \\ \hat{a} =_{\hat{L}} \hat{a'} \text{ and } b \prec_L b' \end{array} \right\} \Rightarrow^1 \hat{a} \diamond_L b \prec_L \hat{a'} \diamond_L b' \Rightarrow^2 (4.21) \text{ holds;}$$

$$\hat{a} =_{\hat{L}} \hat{a'} \text{ and } b =_L b' \left\{ \begin{array}{l} \Rightarrow^3 \}_1 \prec_B \}'_1 \text{ and } \}_2 \preceq_B \}'_2 \\ \Rightarrow^4 \}_1 \preceq_B \}'_1 \text{ and } \}_2 \prec_B \}'_2 \end{array} \right\} \Rightarrow^{1,5}$$

$$\hat{a} \diamond_L b = \hat{a'} \diamond_L b' \text{ and } \}_1 \diamond_B \}_2 \prec_B \}'_1 \diamond_B \}'_2 \Rightarrow^2 (4.21) \text{ holds}$$

where $\Rightarrow^1$ *comes from strict monotonicity of* $\diamond_L$ *in Definition 4.59 on the previous page,* $\Rightarrow^2$ *comes from the product of lattices (i.e., lexicographic product),* $\Rightarrow^3$ *comes from* (4.24), $\Rightarrow^4$ *comes from* (4.25) *and* $\Rightarrow^5$ *comes from strict monotonicity of* $\diamond_B$ *in Definition 4.59.*

□

EXAMPLE 4.62 *Let Integer,* $\Re$, *Bool and Set Integer and let also* $\Re^2 = \langle \Re, \Re \rangle$. *Then we define, for all* $i_1, i_2 \in$ *Integer,* $x_1, x_2, y_1, y_2 \in \Re$, $b_1, b_2 \in$ *Bool and* $s_1, s_2 \in$ *Set Integer such that* $i_1 \preceq_{Integer} i_2$, $x_1 \preceq_\Re x_2$, $y_1 \preceq_\Re y_2$, $b_1 \preceq_{Bool} b_2$ *and* $s_1 \preceq_{Set\ Integer} s_2$,

$$\begin{aligned} \hat{i_1} \diamond_{Integer} i_2 &= i_2 - i_1, \\ \hat{x_1} \diamond_\Re x_2 &= x_2 - x_1, \\ \hat{b_1} \diamond_{Bool} b_2 &= 0.0 \text{ if } b_1 = b_2 \text{ and } 1.0 \text{ otherwise}, \\ \widehat{\langle x_1, y_1 \rangle} \diamond_{\Re^2} \langle x_2, y_2 \rangle &= +\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}, \\ \hat{s_1} \diamond_{Set\ Integer} s_2 &= \#(s_2) - \#(s_1). \end{aligned}$$

Let $X = \{i, i', x, y, s, b\}$ *and suppose* $i, i' \in V_{Integer}$, $x \in V_\Re$, $y \in V_{\Re^2}$, $s \in V_{Set\ Integer}$, $b \in V_{Bool}$ *and*

$$\begin{aligned} c_1 = i \sqsubseteq [\mathbf{1}, \mathbf{4}], \qquad & c_2 = i' \sqsubseteq [\mathbf{1}, \mathbf{4}), \\ c_3 = x \sqsubseteq (\mathbf{3.5}, \mathbf{5.7}), \qquad & c_4 = y \sqsubseteq [(\mathbf{2.0}, \mathbf{3.0}), (\mathbf{3.4}, \mathbf{5.6})], \\ c_5 = s \sqsubseteq [\{\}, \{\mathbf{3}, \mathbf{4}, \mathbf{5}\}) \quad & \text{and } c_6 = b \sqsubseteq [\mathit{false}, \mathit{true}]. \end{aligned}$$

Then

$$\begin{aligned} & \mathit{precision}_{Integer}(c_1) = (3.0, 2), \qquad \mathit{precision}_{Integer}(c_2) = (2.0, 2), \\ & \mathit{precision}_\Re(c_3) = (2.2, 0), \qquad \mathit{precision}_{\Re^2}(c_4) = (2.95, 2), \\ & \mathit{precision}_{Set\ Integer}(c_5) = (3.0, 1), \qquad \mathit{precision}_{Bool}(c_6) = (1.0, 2) \end{aligned}$$

Observe that $\mathit{precision}_{Integer}(c_2)$ *is* $(2.0, 2)$ *instead of* $(3.0, 1)$ *since the constraint* $i' \sqsubseteq [\mathbf{1}, \mathbf{4})$ *is equivalent in the domain* $\mathcal{C}^X$ *(see Section 4.4.4) to the constraint* $i' \sqsubseteq [\mathbf{1}, \mathbf{3}]$, *and the precision is then computed with respect to this last constraint.*

The binary operators used in this example, that is, $-$ and $+$ as well as the unary operators $\#$ and squared need to be complete functions over the whole computation domain and therefore need to include the lifted bounds. The unary operator square root must be defined over the non-negative real domain, including the lifted upper bound.

DEFINITION 4.63 *(Precision of a store) We define the precision of a consistent constraint store $S \in \mathcal{SS}^X$ as*

$$precision(S) = \sum_{c \in S, c \in \mathcal{CC}_L^X \text{ for some } L \in \mathcal{L}} precision_L(c)$$

where the sum in $\Re\mathcal{I}$ is defined as $(a_1, a_2) + (b_1, b_2) = (a_1 + b_1, a_2 + b_2)$.

Therefore, the precision of a consistent simple stable constraint store S is the sum of the precisions of each of its elements.

EXAMPLE 4.64 *Suppose that $S = \{c_1, c_2, c_3, c_4, c_5, c_6\} \in \mathcal{SS}^X$, where c_i for $1 \leq i \leq 6$ is as defined in Example 4.62 on the facing page, and $X = \{i, i', x, y, s, b\}$. Then*

$$precision(S) = \sum_{1 \leq i \leq 6} precision_L(c_i) = (14.15, 9).$$

PROPOSITION 4.65 *Suppose $S, S' \in \mathcal{SS}^X$ are consistent stores where $S \prec_s S'$. Then $precision(S) <_{\Re\mathcal{I}} precision(S')$.*

PROOF 4.66 *By hypothesis $S \prec_s S'$ and then, by Definition 4.33 on page 100,*

$$\forall x \in X \,.\, c_x \preceq_{\mathcal{CC}_L^X} c'_x \text{ and } \exists y \in X \,.\, c_y \prec_{\mathcal{CC}_L^X} c'_y$$

where, for all $x \in X$ and $x \in V_L$ (for some $L \in \mathcal{L}$), c_x and c'_x are the simple consistent constraints in S and S' respectively. Then, by Proposition 4.60 on page 111,

$$\forall x \in X \,.\, precision_L(c_x) \leq_{\Re\mathcal{I}} precision_L(c'_x) \text{ and}$$
$$\exists y \in X \,.\, precision_L(c_y) <_{\Re\mathcal{I}} precision_L(c'_y).$$

Since the sum in $\Re\mathcal{I}$ is defined monotonically as $(a_1, a_2)+(b_1, b_2) = (a_1+b_1, a_2+b_2)$, then

$$\sum_{x \in X, c_x \in S, c_x \in \mathcal{CC}_L^X} precision_L(c_x) <_{\Re\mathcal{I}} \sum_{x \in X, c'_x \in S', c'_x \in \mathcal{CC}_L^X} precision_L(c'_x).$$

Thus, by Definition 4.63,

$$precision(S) <_{\Re\mathcal{I}} precision(S').$$

□

By defining a computable[3] bound $\varepsilon \in \Re^+ \cup \{0.0\}$, we can check if the precision of the simple constraints in a store S were reduced by a significant amount in the stabilisation process (step 4 in the operational schema for $solve(C, S)$). If the change is large enough then the propagation procedure continues. Otherwise the set of simple constraints in the store S is considered a "good enough" solution and the procedure terminates. This "solution" is an approximation to the concept of solution shown in Definition 4.46. The case $S = S'$ is equivalent to the case[4] $precision(S') - precision(S) = (0.0, 0)$ and the case $precision(S') - precision(S) = (0.0, \phi)$ with $\phi > 0$ corresponds, to a situation in which only a propagation or change in the brackets of the ranges associated to the constraints in S was done.

The bound ε is user or system defined. To use $precision/1$ and ε, we define a new operational schema for $solve_\varepsilon(C, S)$ which is the same as that for $solve(C, S)$ apart from step (5) which is replaced by:

$$(5^\star)\ \textit{until } S \textit{ is inconsistent or } precision(S') - precision(S) \leq_{\Re\mathcal{I}} (\varepsilon, 0).$$

THEOREM 4.67 *(Termination) Let $C \in \wp_f(\mathcal{C}^X)$ and $S \in \mathcal{SS}^X$. If $\varepsilon > 0.0$ then the operational schema for $solve_\varepsilon(C, S)$ terminates.*

PROOF 4.68 *Suppose that S_0 is the initial value of S. If S_0 is inconsistent, then the test in step (0) fails and the procedure terminates without entering the repeat loop. Suppose S_0 is consistent so that there is at least one iteration of the repeat loop. For $i \geq 1$, let S_i denote the value of constraint store S at the end of the i'th iteration of the repeat loop. If for any i, S_i is inconsistent, then the test in step ($5^\star$) is true and the procedure terminates. Suppose now that, for $i \geq 1$, S_i is consistent. In the first iteration we have, by step (4), $S_0 \cup C' \mapsto S_1$ so that, by Proposition 4.35 on page 100, $S_1 \preceq_s S_0$. Then, if $S_1 = S_0$ the procedure terminates since $precision(S_1) =_{\Re\mathcal{I}} precision(S_0)$ so that the test in step ($5^\star$) is true. Otherwise, $S_1 \prec_s S_0$ and thus, by Proposition 4.65 $precision(S_1) <_{\Re\mathcal{I}} precision(S_0)$. Then, $precision(S_1) <_{\Re\mathcal{I}} \top_{\Re\mathcal{I}}$ so that, for some constant k, we have*

$$precision(S_1) <_{\Re\mathcal{I}} (k \times \varepsilon, 0).$$

We will show by induction on $i \geq 1$ that

$$precision(S_1) - precision(S_i) \geq_{\Re\mathcal{I}} \big((i-1) \times \varepsilon, 0\big).$$

It follows that $i \leq k+1$ and the procedure terminates after no more than $k+1$ iterations.

The base case, when $i = 1$ is obvious. Suppose next that $i > 1$ and that the hypothesis holds for $i - 1$. If there is an i iteration, then, by the condition in step ($5^\star$) of the repeat loop,

$$precision(S_{i-1}) - precision(S_i) >_{\Re\mathcal{I}} (\varepsilon, 0).$$

[3] That is, representable in the machine which is being used - the computation machine.

[4] The difference in $\Re\mathcal{I}$ is defined as $(a_1, a_2) - (b_1, b_2) = (a_1 - b_1, a_2 - b_2)$.

However, by the inductive hypothesis,

$$precision(S_1) - precision(S_{i-1}) \geq_{\Re\mathcal{I}} \big((i-2) \times \varepsilon, 0\big)$$

so that

$$\begin{aligned} precision(S_1) - precision(S_i) &\geq_{\Re\mathcal{I}} (\varepsilon, 0) + \big((i-2) \times \varepsilon, 0\big) \\ &= \big((i-1) \times \varepsilon, 0\big). \end{aligned}$$

□

DEFINITION 4.69 *(Approximate solution) Let $C \in \wp_f(\mathcal{C}^X)$ and $S \in \mathcal{SS}^X$. Let also R be a solution for $C \cup S$ and $(\delta, \phi) \in \Re\mathcal{I}$. An* approximate solution via (δ, ϕ) for $C \cup S$ *is a simple stable constraint store for X denoted as $R_{(\delta,\phi)}$ where,*

$$precision(R_{(\delta,\phi)}) - precision(R) \leq_{\Re\mathcal{I}} (\delta, \phi) \quad and \quad R \preceq_s R_{(\delta,\phi)}.$$

The number of iterations of the operational schema depends, for infinite domains, on the value of ε. In these cases, the final solution for $solve_\varepsilon(C, S)$ is an approximate solution for $C \cup S$.

THEOREM 4.70 *(Extended correctness) Let $C \in \wp_f(\mathcal{C}^X)$ and $S \in \mathcal{SS}^X$. If at least one solution R for $C \cup S$ exists, a terminating execution of the operational schema for $solve_\varepsilon(C, S)$ computes in S an approximate solution for $C \cup S$.*

PROOF 4.71 *Suppose the procedure terminates after k iterations with S_ε the value of the store S. It has already been shown in the proof of Theorem 4.56 in line* (4.16) *that*

$$R \preceq_s S_\varepsilon,$$

so that by Proposition 4.65 on page 113

$$precision(R) \leq_{\Re\mathcal{I}} precision(S_\varepsilon),$$

and thus, for some $(\delta, \phi) \in \Re\mathcal{I}$,

$$precision(S_\varepsilon) - precision(R) = (\delta, \phi).$$

Then, by Definition 4.69, S_ε is an approximate solution for $C \cup S$.
□

Also, in these cases, the approximate solution is dependent on the value of ε in the sense that lower ε is, closer the approximate solution to the solution is.

THEOREM 4.72 *Let R be a solution for $C \cup S$ where $C \in \wp_f(\mathcal{C}^X)$ and $S \in \mathcal{SS}^X$. Suppose that S_{ε_1} and S_{ε_2} are the approximate solutions computed by the operational schema for $solve_{\varepsilon_1}(C, S)$ and $solve_{\varepsilon_2}(C, S)$, respectively. Then, if $0.0 \leq \varepsilon_1 \leq \varepsilon_2$,*

$$R \preceq_s S_{\varepsilon_1} \preceq_s S_{\varepsilon_2}.$$

PROOF 4.73 *In the previous proof we have* $R \preceq_s S_\varepsilon$ *for any* $\varepsilon \geq 0.0$. *Therefore* $R \preceq_s S_{\varepsilon_1}$ *and* $R \preceq_s S_{\varepsilon_2}$ *Thus, we just have to show that* $S_{\varepsilon_1} \preceq_s S_{\varepsilon_2}$.

Suppose the procedures $solve_{\varepsilon_2}(C, S)$ *and* $solve_{\varepsilon_1}(C, S)$ *terminate after* k_2 *and* k_1 *iterations respectively. Therefore, the check, in line (5***), for the repeat loop for iterations from 1 to* k_2 *must also succeed for* ε_1. *Thus* $k_1 \geq k_2$.

Suppose that S_i *is the value of* S *at the end of the i'th iteration of the repeat loop* $(1 \leq i \leq k_1)$ *(so that* $S_{k_1} = S_{\varepsilon_1}$ *and* $S_{k_2} = S_{\varepsilon_2}$*). We show, by induction on i, where* $k_2 \leq i \leq k_1$, *that*

$$S_i \preceq_s S_{\varepsilon_2}. \tag{4.26}$$

The base case when $i = k_2$ *is obvious. For the inductive step, suppose that* $i > k_2$ *and assume that* $S_{i-1} \preceq_s S_{\varepsilon_2}$. *By step (4) of the extended operational schema, we have*

$$S_{i-1} \cup C' \mapsto S_i \tag{4.27}$$

at the end of the i'th iteration of the repeat loop. Thus, by Proposition 4.35 on page 100, $S_i \preceq_s S_{i-1}$ *so that* (4.26) *holds. Therefore, letting* $i = k_1$ *we obtain the required result*

$$S_{\varepsilon_1} \preceq_s S_{\varepsilon_2}.$$

□

The *precision* map and the bound ε allow direct and transparent control over the accuracy of the results. For example, we could define $\varepsilon = 10^{-8}$ for reals. Together, the *precision* map and the bound ε provide a concept of graded solutions to a constraint problem as well as a concept of distance to the correct solution: the higher the bound ε, then the further away (from the correct solution) is the approximate solution. The set of approximate solutions are ordered by the ordering relation on stores shown in Definition 4.33.

The termination of our algorithm consists of measuring the propagation by the difference of the precision of stores. This concept is generic and, to our knowledge, novel. A naive approach, only on reals, consists of associating a precision parameter with the real domain to limit the number of times that the domain of a variable can be refined (Sidebottom and Havens, 1992).

4.6 Instances of Our Framework

4.6.1 Classical Domains

Most classical constraint domains are lattices. For instances, the *Integer*, $\Re$, *Bool*, and *Set* L (for any domain $L \in \mathcal{L}$) domains have been used throughout the chapter to illustrate the concepts as they were defined. Here are examples of constraint intersection in the interval domain over these domains:

(1) $i \sqsubseteq \mathbf{[1,8)} \cap_L i \sqsubseteq \mathbf{(0,5]} = i \sqsubseteq \mathbf{[1,5]}$;
(2) $r \sqsubseteq \mathbf{[1.12,5.67)} \cap_L r \sqsubseteq \mathbf{[2.34,5.95)} = r \sqsubseteq \mathbf{[2.34,5.67)}$;
(3) $b \sqsubseteq (\mathit{false}, \mathit{true}] \cap_L b \sqsubseteq [\mathit{false}, \mathit{true}] = b \sqsubseteq (\mathit{false}, \mathit{true}]$.

Also suppose that $L_1 \in \mathcal{L}$. Then, *Set* L_1 is a lattice over which it is possible to solve set constraints. For example, consider the following store:

$$\begin{aligned} S \;=\; \{\; & s ::' \mathit{Set\ Integer}, \\ & s \sqsubseteq [\mathbf{\{1\},\{1,2,3,4\}}], \\ & s \sqsubseteq [\mathbf{\{3\},\{1,2,3,5\}}] \;\} \end{aligned}$$

By applying $\cap_L$ twice, constraints in S are solved as follows[5]:

$$\begin{aligned} s \sqsubseteq [\emptyset, \top_{\mathit{Set\ Integer}}] \;\cap_L\; s \sqsubseteq [\mathbf{\{1\},\{1,2,3,4\}}] \;&=\; s \sqsubseteq [\mathbf{\{1\},\{1,2,3,4\}}] \\ s \sqsubseteq [\mathbf{\{1\},\{1,2,3,4\}}] \;\cap_L\; s \sqsubseteq [\mathbf{\{3\},\{1,2,3,5\}}] \;&=\; s \sqsubseteq [\mathbf{\{1,3\},\{1,2,3\}}] \end{aligned}$$

4.6.2 Computation Domains: More Examples

Here we provide further examples of (possibly user-defined) computation domains and indicate how they may be used.

Binary strings.

The domain of binary strings $\mathcal{B}$ is the set of all sequences (possibly infinite) of zeros and ones together with $\top_{\mathcal{B}}$. The bottom element $\bot_{\mathcal{B}}$ is the empty sequence. For all $b_1, b_2 \in \mathcal{B}$, we define $b_1 \preceq_{\mathcal{B}} b_2$ if and only if b_1 is a prefix (finite initial substring) of b_2. Therefore $glb_{\mathcal{B}}(b_1, b_2)$ is the largest common prefix of b_1 and b_2 (e.g. $glb_{\mathcal{B}}(00010, 00111) = 00$) and $lub_{\mathcal{B}}(b_1, b_2)$ is $\top_{\mathcal{B}}$ if $b_1 \not\sim b_2$, b_2 if $b_1 \preceq_{\mathcal{B}} b_2$ and b_1 if $b_2 \preceq_{\mathcal{B}} b_1$. Therefore

$$(\mathcal{B}, \preceq_{\mathcal{B}}, glb_{\mathcal{B}}, lub_{\mathcal{B}}, \bot_{\mathcal{B}}, \top_{\mathcal{B}})$$

is a lattice. Thus, with $\mathcal{B}$ as the computation domain and $+/2$ a constraint operator that concatenates two binary strings, the following interval constraints for $x, y \in V_{\mathcal{B}}$

$$x, y ::' \mathcal{B}, \qquad x \sqsubseteq [001\overline{+}min(y), \top_{\mathcal{B}}]$$

constrain x to have values in the set of binary strings that start with the sequence 001.

Non-negative integers ordered by division.

Let $\mathcal{N}_d$ denote the set of non-negative integers partially ordered by division: for all $n, m \in \mathcal{N}_d$, $m \preceq_{\mathcal{N}_d} n$ if and only if $\exists k \in \mathcal{N}_d$ such that $km = n$ (that is, m divides n). Then

$$(\mathcal{N}_d, \preceq_{\mathcal{N}_d}, gcd, lcm, 1, 0)$$

is a lattice where *gcd* denotes the greatest common divisor and *lcm* the least common multiple. Thus with $\mathcal{N}_d$ as the computation domain we have:

$$x \sqsubseteq \big[2, 24\big] \;\cap_L x \sqsubseteq [3, 36] \quad = \quad x \sqsubseteq [6, 12].$$

[5] Observe that $s ::' \mathit{Set\ Integer}$ is the constraint $s \sqsubseteq [\emptyset, \top_{\mathit{Set\ Integer}}]$.

Numeric intervals

We consider *Interv* as the domain of the (possibly continuous) numeric intervals. We define $it_1 \preceq_{Interv} it_2$ if and only if $it_1 \subseteq it_2$ (i.e., it_1 is a subinterval of it_2). Thus glb_{Interv} and lub_{Interv} are the intersection and union of intervals respectively. Our framework solves constraints for the *Interv* computation domain as follows:

$$i \sqsubseteq [[5,6],[2,10)] \ \cap_L \ i \sqsubseteq [(7,9],[4,15]] \ = \ i \sqsubseteq [[5,6] \cup (7,9],[4,10)]$$

where $[4,10) = [2,10) \cap [4,15)$. Observe also that $[[5,6] \cup (7,9] \in \overline{Interv^s}$ whereas $[4,10)] \in Interv^s$.

4.6.3 Combinations of Domains

Our lattice-based framework allows for new computation domains to be constructed from previously defined domains.

Product of domains

As already observed, the direct and lexicographic products of lattices are lattices. For example, consider the lattice *Integer*.
(1) A point in a plane may be defined by its Cartesian coordinates using the direct product $Point = \langle Integer, Integer \rangle$.
(2) A rectangle[6] can be defined by two points in a plane: its lower left corner and its upper right corner. Let *Rect* be the direct product $\langle Point, Point \rangle$.

Interval constraints can be declared directly on these domains. For example, consider $re \in V_{Rect}$, then

$$re \sqsubseteq [\langle \langle 2,2 \rangle, \langle 5,5 \rangle \rangle, \langle \langle 4,4 \rangle, \langle 7,7 \rangle \rangle]$$

constrains the rectangle *re* to have its lower left corner in the plane $\langle 2,2 \rangle \times \langle 4,4 \rangle$ and its upper right corner in the plane $\langle 5,5 \rangle \times \langle 7,7 \rangle$. Thus the rectangle $\langle \langle 3,3 \rangle, \langle 6,6 \rangle \rangle$ satisfies this constraint.

Sum of Domains

As already declared, the linear sum of $n > 1$ lattices is also a lattice.

As an example, consider the lattice *AtoF* containing all the (uppercase) alphabetic characters between 'A' and 'F' with the usual alphabetical ordering and the lattice *0to9* containing the numeric characters from '0' to '9' with the ordering '0'<'1'<...<'8'<'9'. Then the lattice of hexadecimal digits *0toF* can be defined as the lattice $0to9 \uplus AtoF$. Now, it is possible to constrain variables to have values in such a domain. For example a code of four hexadecimal digits can be initially represented by four variables $h_1h_2h_3h_4$, belonging to V_{0toF}, constrained initially by a type constraint as

$$h_1, h_2, h_3, h_4 ::' \ 0toF$$

[6] We assume a rectangle that has its base parallel to the x axis.

Note that this is equivalent to the constraints

$$h_1 \sqsubseteq [\text{'0'},\text{'F'}], h_2 \sqsubseteq [\text{'0'},\text{'F'}], h_3 \sqsubseteq [\text{'0'},\text{'F'}], h_4 \sqsubseteq [\text{'0'},\text{'F'}].$$

4.7 Related Work

4.7.1 Interval Reasoning

In Section 2.6.5 we discussed the integration of constraints and interval arithmetic. The indexical approach of clp(FD) to constraint propagation, from which our framework is derived, is based on interval arithmetic. Note that, interval arithmetic has been applied to constraint satisfaction problems (Benhamou, 1995; Lee and van Emden, 1993; Older, 1989) over numeric domains (Benhamou and Older, 1997) and, in particular, to floating point numbers on relational programming (Cleary, 1987). In this latter application, interval computations are used to approximate a computed real number. This concept of approximation which works well on numeric domains does not generalise since it assumes that the closest value smaller (resp. higher) than any computed value is computable. Thus, for our framework, we have defined a new system of approximation for infinite domains that ensures that the constraint solver only has to deal with computable values.

Older and Vellino in (Older and Vellino, 1993) present a lattice-theoretic semantics for numeric interval constraints Here, constraint interval arithmetic is described as a classical numerical computation technique as well as a proof technique that allows to prove the non-existence of solutions. The framework proposed in (Older and Vellino, 1993) aims to capture the properties of both the primitive interval operations and the constraint propagation networks created from them. Based on lattice theory, some analogies with respect to our proposal can be detected: (1) the computation domain has a lattice structure and is constructed from the bounds of the intervals (2) the theory can be applied with infinite precision (i.e., without approximating a real to a floating point number) although only on reals; (3) the operators are assumed to maintain properties over the computation domain that are also maintained by our constraint operators (e.g., monotonicity); (4) the propagation process is based on a fixed point semantics. In spite of the similarities, there are a number of aspects that made this approach very different from our proposal: (1) The framework is developed exclusively for numeric domains and then assumes all the basic operations on reals as the primitive operations; as consequence the initial intervals of the variables are always numeric and the solutions range only over numeric intervals; (2) we provide a control mechanism, at the user level, for the propagation by allowing the constraint operators to be defined directly on the bounds of the interval; in this sense, Older and Vellino do not treat the issue of the transparency of their theory; (3) the theory proposed in (Older and Vellino, 1993) is "quite abstract and therefore somewhat remote from actual implementations", whereas our theory can be directly implemented; (4) the theory described in (Older and Vellino, 1993) does not complete the solving, as we do in Chapter 6, so that to obtain all the solutions simply relies on the backtracking mechanism of Prolog; moreover solver

cooperation is not treated at all in (Older and Vellino, 1993) and implementation issues are neglected (see Chapters 5 and 7, respectively).

4.7.2 Generic Frameworks

In this section we discuss several proposals for finding general principles for constraint satisfaction.

Constraint Propagation from Chaotic Iterations (Apt, 1999) proposes a framework for constraint propagation that is based on chaotic iteration (CI) algorithms for partially ordered domains. The key idea is that most constraint propagation algorithms presented in the literature can be expressed as direct instances of these algorithms. Apt also shows in (Apt, 1999) how new constraint propagation algorithms for some domains (i.e., those that hold a property called the finite chain property) may be obtained as instances of this framework.

In (Fernández and Hill, 1999c) we presented a preliminar version of our propagation algorithm presented in this chapter. The propagation frameworks described in (Fernández and Hill, 1999c) and (Apt, 1999) appear to have some analogies; in particular, both consider schemas for constraint propagation procedures for partially ordered domains, finite or infinite. In (Fernández and Hill, 1999b) we established the relationship between these approaches so that the properties of the algorithms shown in each of them can be generalised to the other. We presented a modified version of the chaotic iteration algorithm based on one presented in (Apt, 1999). This modified algorithm relaxes the requirement that domains satisfy the finite chain property and can be used for any domain that is a lattice. The chaotic iteration approach of (Apt, 1999) is then used to derive a constraint propagation algorithm from the interval-lattice based constraint propagation framework described in (Fernández and Hill, 1999c). Then, by establishing the connection between these two theories, we were able to deduce correctness and termination results for algorithms based on our operational schema shown in Section 4.5 directly from the results about soundness and termination for the chaotic iteration algorithm.

The main similarities between both approaches were also shown by identifying the common elements in the algorithms described in (Apt, 1999) and in (Fernández and Hill, 1999c). It was shown that the process of constraint propagation in our operational schema can be viewed as a process of function evaluation in the chaotic iteration algorithm of Apt.

Two main differences between the approach in (Apt, 1999) and its application to the interval constraints procedure presented here were shown. In (Apt, 1999), a formalism for solving a constraint satisfaction problem was presented. It was then shown how the chaotic iteration algorithm can be applied to such a formalism. However, we used the chaotic iteration algorithm directly rather than via this formalism. The reason was that our interval constraints do not correspond to the constraints as defined there. These represented sets of possible solutions whereas our interval constraints embody more; the intended dependence between the variables for the constraint propagation. Thus,

our interval constraints correspond to functions that are used in the chaotic iteration algorithm. The second difference concerns the chaotic iteration algorithm itself. Since our domains (such as the reals) do not necessarily possess the finite chain property (which is assumed in the domains treated in (Apt, 1999)), to ensure termination we changed the main test in the chaotic algorithm to a test where domain elements are approximated by values in the domain of non-negative integers.

Constraint Solving on Semirings Bistarelli et al. (Bistarelli et al., 1995) described, for finite domains, a general framework for constraint solving over CSPs in which a certain level of preference (i.e., cost or degree) is associated to each tuple of values of the domain of the constrained variables. The framework is based on a finite semiring structure where the set of the semiring specifies the values to be associated to each tuple of values of the domain of the constrained variables. It was also shown how to combine constraints from one of the operations of the semiring ($\times$). This framework is adequate for classical CSPs, partial CSPs, fuzzy CSPs and weighted CSPs among others.

Later on, (Bistarelli et al., 1997a) extended the CLP formalism in order to handle semiring-based constraint systems and different semantics for a logical language based on this kind of constraint solving were described. The syntax of the semiring CLP programs was also described.

One significant difference with respect to our framework (among others derived from the different ways of solving the constraints) is that Bistarelli et al. require the computational domain to be finite, that is, they do not relax the assumption of a finite domain for the variables whereas our framework allows to work on non-finite domains. As consequence o the termination of the principal algorithm is only guaranteed by forcing the domain of the variables to be finite.

Another difference is that, in passing CLP programs to semiring-based CLP (SCLP) programs, the semantics of CLP have to be modified since in SCLP ground atoms do not necessarily correspond to truth values (as in CLP) but also to elements of the semiring. In our framework, the semantics of CLP in this sense are maintained.

Valued Constraint Satisfaction Problems In (Schiex et al., 1995) a simple algebraic framework, related to over-constrained CSPs was described. Schiex et al. proposed a framework to solve CSPs in which different types of knowledge are taken into account as for example costs, uncertainties, preferences, priorities,etc. This framework relies on a simple structure, a totally ordered commutative monoid (that is, a totally ordered domain plus one operation satisfying some properties), in which the values of the domain are interpreted as costs, or degrees, or probabilities or others. It was observed that this kind of structure is enough to encompass most of the existing CSP extensions in such a way that specific choices of the monoid will then give rise to different instances of the framework.

One of the main differences with respect to our approach is, again, relative to the cardinality of the variable domain. (Schiex et al., 1995) associates each constrained

variable with a finite domain whereas we do not necessarily require it.

More recently, in (Bistarelli et al., 1999), this framework and the semiring-based framework were compared and it was shown that they present the same theoretical expressive power. It was also shown that these settings are mainly focused in solving soft CSPs (CSPs with a confidence value tuple of values associated to a constraint) which is very different to our approach.

Others A generic arc consistency algorithm, called AC-5, is also presented in (Van Hentenryck et al., 1992). The genericity of the AC-5 algorithm lies in that it is parametrised on two specified procedures whose implementation is left open. The algorithm, by the proper implementation of these procedures, can be instantiated to devise the well known algorithms AC-3 (Mackworth, 1977) and AC-4 (Mohr and Henderson, 1986). Despite of this claim of genericity, in fact the algorithm assumes a total ordering on the computation domain and thus it is a very limited form of genericity. In fact, of the classic domains, the algorithm is only valid for the integer and the Boolean domain. Moreover, the algorithm itself provides a propagation procedure for the kernel of the system CHIP for finite domains.

Also (Hong and Ratschan, 1995) describes a curious constraint system that provides a possibility for instantiating a CLP language with an arbitrary domain. This paper presents the RISC-CLP(Tree(Δ)) system that provides an interface on several levels of abstractions to user-defined constraint domains. The alleged advantage is that the "user can first implement a simple but inefficient prototype and then refine it to more complicated but efficient implementations". This system offers improved genericity but at expenses of declarativity since the constraint solver for each user-defined domain has to be defined as a C++ class. The resulting programs are an amalgamation of C++ and Prolog code which is a long way from the declarativity of the CLP paradigm.

Another generic proposal is that of (Le Provost and Wallace, 1993) that shows that propagation is domain independent by proposing a generalisation of propagation, called *generalised propagation (GP)*, which can be applied on arbitrary computation domains. In contrast to our proposal, the constraints are not defined generically here but the main idea behind this approach is to use any available constraint over any computation domain X to express restrictions on problem variables. An important drawback of GP is that termination of the search for answers to a propagation constraint is not guaranteed and the entire responsibility for ensuring termination is given to the programmer. The main instance of GP was proved to be that over the Herbrand Universe i.e., GP(HU).

4.8 Concluding Remarks

In this chapter we have defined a framework for interval constraint propagation over lattices and illustrated with many examples the versatility and expressivity of this approach. For maximum generality and allowing for any lattice, finite or infinite, discrete or continuous, we have constructed the interval domains used for defining and

propagating the constraints in several stages; each stage taking advantage of the lattice structures inherited from the underlying computation domains on which the interval domains are built.

Thus, we first defined and added the bracket domain B to each computation domain L to create the right bounded domain L^s for open and closed (right) bounds **a)** and **a]** for the intervals. We then defined a symmetric mirror domain $\overline{L^s}$ so as to allow for the left bounds of intervals **(a** and **[a**. These bounds were then combined using the direct product of lattices to form the range elements $r = \{_{\mathbf{1}}\mathbf{a}, \mathbf{b}\}_{\mathbf{2}}$ of an interval domain R^s_L. Finally, we added the variable to be constrained to the given range to form the interval constraint $x \sqsubseteq r$.

This summarises the construction of a simple interval constraint. When defining all the elements of the bounded computation domain L^b, we introduced two additional constructs. One, which generalises an idea from (Codognet and Diaz, 1996a), was indexicals $max(x), val(x)$, for the right bounded domain and $min(x), \overline{val(x)}$, for the left bounded domain. These provide necessary links between the ranges for the constrained variables and give the user useful transparent control over the constraint propagation. The other was that of an operator $\circ_L$ which maps a domain constructed from several, possibly distinct, computation domains $L_1, \ldots, L_n$ to another, possibly different, co-domain L. This, combined with the indexicals, allows a one-way communication from the domains $L^b_1, \ldots, L^b_n$ to the domain L^b.

We have presented an operational schema for propagating these constraints and proved it correct. In the case of the non-finite domains, termination of the procedure can only be guaranteed by letting the solver return an approximation to the correct result. An idea from (Sidebottom and Havens, 1992) for controlling accuracy in the processing of disjoint intervals over the reals was adapted for our lattice domains. The special operator $precision_L$ that maps the domain elements to elements of the combined domain $(\Re^+, Integer)$ to check the variations of the intervals (including variations just in the brackets) and a limit element $\varepsilon \in \Re^+ \cup \{0.0\}$ that controls the degree of the approximation were introduced. The basic operational schema was then adapted so as to check, using these precision and limit constructs, for just an approximation to the fix-point. With this modification of the schema, we proved that such a procedure terminates with an approximate solution.

Observe that the framework, being applicable to any lattice, provides support for all the existing practical domains in CLP (e.g. reals, integers, sets and Booleans). Moreover, by using lattice combinators, new compound domains and their solvers can easily be obtained from previously defined domains such as these.

We have also proved that each constraint defined over the generic structure guarantees the monotonicity of the constraint solving.

4.9 Contributions

We have proposed a generalisation of the indexical model. The novelty of our proposal, which is also novel for the FD, for dealing with interval constraints arises mainly from

1. the generic construction of an interval-lattice structure over which the constraints are propagated and that is independent of the computation domain and

2. the definition of the constraint operators over multiple domains to generate one-way channels (via the definition of constraint operators defined for multiple domains) through which information can flow between domains;

Our generic interval propagation schema also provides other contributions that are enumerated below:

- Our theoretical framework provides a new mechanism for checking *a priori* if the interval constraints defined on indexical-based systems are monotonic.

- Many existing constraints systems can be seen as instances of our interval propagation schema. This means that our schema can be used to explain the procedural behaviour of these instances and thus can guarantee key properties such as termination and correctness. In this sense, our theoretical setting also shows that it is the lattice structure of constraint domains that lies at the heart of many propagation-based constraint programming systems

- Our framework provides an alternative to the generic and transparent CHR approach.

Chapter 5

Interval Solver Cooperation

Un pour tous et tous pour un.

Les Trois Mousquetaires (1844)
Alexandre Dumas, 1802-70

5.1 Introduction and Motivation

In Chapter 4, we have proposed an alternative to the CHR approach that is based on the indexical approach of clp(FD) (Codognet and Diaz, 1996a) which, as shown in Chapter 3, is known to be highly efficient. In this proposal, we have defined a framework for interval constraint propagation on any domain with a lattice structure. Thus, this framework is valid on all the classical domains such as Booleans, reals, finite ranges of integers and sets as well as new specialised domains designed for specific applications.

However, in practice, constraints are often not specific to any given domain. Therefore the formulation of real problems has to be artificially adapted to a domain that is supported by the system. Many problems are most naturally expressed using heterogeneous constraints, involving more than one domain. For example, consider a community of people. To write a constraint that determines the set of people that are taller than two meters cannot be directly coded in most existing CLP languages. The main exception to this is the CHR language (Frühwirth, 1998) that allows for user-defined domains.

In this chapter we generalise our interval propagation framework described in the previous chapter to allow the cooperation of solvers (possibly defined on different domains). We start from the foundations established in Chapter 4 where we have shown that when defining all the elements of the bounded computation domain L^b, we allowed the definition of a set of operators $\circ_L$ mapping a domain constructed from several, possibly distinct, computation domains $L_1^s, \ldots, L_n^s$ to another, possibly different, co-domain

L^s. This, combined with the indexicals, allows a one-way communication from the domains $L^b_1, \ldots, L^b_n$ to the domain L^b.

Full solver cooperation requires two-way communication between domains. Thus we define a high level constraint as a relation over a domain constructed from a set of, possibly distinct, computation domains and show how, with this construct, we can solve non-trivial problems (e.g., we show how, at the user level, we can define complex constraints such as *reified constraints* that reflect the validity of a constraint (over any domain) into a Boolean variable -see Section 2.7).

EXAMPLE 5.1 *Consider the direct product domain*

$$\text{community} = \text{string} \times \Re$$

(where string *is a set of names) that defines a community of people ordered lexicographically by their name (first argument) and height (second argument). Assume that a high level constraint* $>'/2$ *on the domain* community $\times \Re$ *is also defined, for* name $\in$ string *and* height, limit $\in \Re$*, as*

$$(\text{name}, \text{height}) >' \text{limit} \Leftrightarrow \text{height} >_{\Re} \text{limit}.$$

Then, the constraint

$$person >' 2.00,$$

where $person \in$ community*, determines the set of people higher than two meters.*

Moreover, as the basic framework enables new domains to be constructed from existing domains by means of lattice combinators (see Section 4.6.3), the generalised setting allows constraint propagation to go from the basic domains to the combined domains and vice-versa. A direct consequence is that the expressiveness of our framework is improved.

5.1.1 Chapter Structure

In Section 5.2 we define the concept of *high level constraint*. Observe then that the constraint operators were defined on multiple domains (see Definition 4.10 on page 88) and that they provide a useful one-way channel of communication from the computation domains for the arguments of the operator to the (possibly compound) computation domain in its range and that, in this chapter, we define high level constraints which enable the propagation of information between the domains to occur in any direction and allow for real cooperation between the solvers. So as to distinguish the interval constraints defined and studied in the previous chapter from the high level constraints defined in 5.2, we call constraints of the form $x \sqsubseteq r$ *primitive constraints*. In Section 5.3, we give non-trivial examples of high level constraints that allow for propagation (and, hence, cooperation) over combined domains. Then, in Section 5.4 we provide another example to show that propagation can also be done among non-related domains. Section 5.5 develops a more practical example (in the biomedicine field) that combines

constraint propagation, solver cooperation and domain combination in order to show the flexibility of the framework. As usual, the chapter terminates with a discussion about related work, the conclusions and some comments about the real contributions of the chapter.

5.2 High Level Constraints

In the same way as in clp(FD) (Codognet and Diaz, 1996a), high level constraints can be defined in terms of other (high level or primitive) interval constraints.

DEFINITION 5.2 *(High level constraint) Suppose that* $\mathcal{L}' = \{L_1, \ldots, L_m\} \subseteq \mathcal{L}$. *Then* $q :: L_1 \times \ldots \times L_m$ *is called an* m-ary constraint relation *for* $\mathcal{L}'$. *Suppose* $x_1 \in V_{L_1}, \ldots, x_m \in V_{L_m}$, *and* $c_1, \ldots, c_n$ *are constraints with constrained variables* $X \supseteq \{x_1, \ldots, x_m\}$. *Then*

$$q(x_1, \ldots, x_m) \Leftrightarrow c_1, \ldots, c_n.$$

is called a high level constraint *over* $\mathcal{L}'$.

However, unlike clp(FD), our framework also allows for the definition of both generic and overloaded constraints. A high level constraint is *generic* for arguments $i_1, \ldots, i_j$ $(1 \leq i_1 < \cdots < i_j \leq m)$ if its definition is independent of the choice of domains $L_{i_1}, \ldots, L_{i_j}$ in $\mathcal{L}$. A constraint is *overloaded* for arguments $i_1, \ldots, i_j$ if it is defined for any $L_{i_1}, \ldots, L_{i_j}$ in $\mathcal{L}_1$ where $\mathcal{L}_1 \subset \mathcal{L}$ and $\#(\mathcal{L}_1) > 1$.

EXAMPLE 5.3 *Consider the following high level 'less-or-equal-than' constraint:*

$$\begin{aligned} x \leq y \Leftrightarrow x \sqsubseteq [\bot_L, & max(y), \\ y \sqsubseteq [& min(x), \top_L]. \end{aligned} \tag{5.1}$$

Then this is generic for both arguments of $\leq$. *Note that each* $L \in \mathcal{L}$ *has (possible lifted) top and bottom elements.*

EXAMPLE 5.4 *Consider the definition of the operators* $-$ *and* $+$ *shown in Example 4.11 and the following definition of a* plus/3 *constraint:*

$$\begin{aligned} plus(x, y, z) \Leftrightarrow x &\sqsubseteq min(z) \overline{-} max(y), max(z) - min(y), \\ y &\sqsubseteq min(z) \overline{-} max(x), max(z) - min(x), \\ z &\sqsubseteq min(x) \overline{+} min(y), max(x) + max(y). \end{aligned}$$

where $x, y, z \in V_L$ *for some* $L \in \mathcal{L}$. *This constraint is overloaded since it is valid for any domain* L *in which operators* $+_L$ *and* $-_L$ *are defined.*

Let $c \Leftrightarrow c_1, \ldots c_n$ be a high level constraint in a set of constraints C. To propagate C with respect to a simple stable constraint store S, we first replace c (in C) by $c_1, \ldots, c_n$. We do this repeatedly until C just contains primitive constraints. Of course, termination of this is not guaranteed and will depend on the definitions of the high-level constraints. As soon as C consists of just primitive constraints, constraint propagation and constraint stabilisation are executed as usual (i.e., as shown in Chapter 4).

EXAMPLE 5.5 *Consider* $\mathcal{L} = \{Set\ \Re, \Re\}$*, the variables* $x, y \in V_{Set\ \Re}$*,* $z, w \in V_{\Re}$ *and the following simple stable constraint stores:*

$$\begin{aligned}
S = \{\ & x \sqsubseteq [\{\mathbf{1.2}, \mathbf{3.0}\}, \{\mathbf{5.6}, \mathbf{1.2}, \mathbf{3.0}, \mathbf{7.4}, \mathbf{9.3}\}),\ z \sqsubseteq (\mathbf{2.0}, \mathbf{15.5}), \\
& y \sqsubseteq [\{\}, \{\mathbf{3.0}, \mathbf{1.2}, \mathbf{7.4}, \mathbf{4.8}\}), \qquad w \sqsubseteq [\mathbf{0.0}, \mathbf{12.0}]\ \}; \\
S' = \{\ & x \sqsubseteq [\{\}, \{\mathbf{3.0}, \mathbf{1.2}, \mathbf{7.4}, \mathbf{4.8}\}), \qquad z \sqsubseteq [\bot_{\Re}, \mathbf{12.0}], \\
& y \sqsubseteq [\{\mathbf{1.2}, \mathbf{3.0}\}, \top_{Set\ \Re}], \qquad w \sqsubseteq (\mathbf{2.0}, \top_{\Re}]\ \}; \\
S'' = \{\ & x \sqsubseteq [\{\mathbf{1.2}, \mathbf{3.0}\}, \{\mathbf{3.0}, \mathbf{1.2}, \mathbf{7.4}\}), \qquad z \sqsubseteq (\mathbf{2.0}, \mathbf{12.0}], \\
& y \sqsubseteq [\{\mathbf{1.2}, \mathbf{3.0}\}, \{\mathbf{3.0}, \mathbf{1.2}, \mathbf{7.4}, \mathbf{4.8}\}), \qquad w \sqsubseteq (\mathbf{2.0}, \mathbf{12.0}]\ \}.
\end{aligned}$$

Consider now the high level constraint $\leq/2$ *as defined in Example 5.3. Then* $\{x \leq y,\ z \leq w\} \rightsquigarrow^S S'$ *and* $S \cup S' \mapsto S''$*.*

EXAMPLE 5.6 *Suppose that* $L \in \{Integer, \Re, Point\}$*, where Point is as defined in Section 4.6.3 on page 118 (i.e.,* $Point = \langle Integer, Integer \rangle$*), that operators* $+$ *and* $-$ *are as defined in Example 4.11 on page 88 where* $+_L$ *and* $-_L$ *have their usual definitions in the integer and real domains (and return the sum and difference of two numbers in* L*) and* $+_{Point}$ *and* $-_{Point}$ *are defined as*

$$\begin{aligned}
\langle a, b\rangle +_{Point} \langle c, d\rangle &= \langle a + c, b + d\rangle, \\
\langle a, b\rangle -_{Point} \langle c, d\rangle &= \langle a - c, b - d\rangle,
\end{aligned}$$

that $x, y, z \in V_{Integer}$*, that* $r, w, t \in V_{\Re}$*, that* $p_1, p_2, p_3 \in V_{Point}$ *and consider the following constraint store:*

$$\begin{aligned}
S = \{& t \sqsubseteq [\mathbf{1.0}, \mathbf{4.0}],\ w \sqsubseteq (\mathbf{0.0}, \mathbf{90.0}], \\
& x \sqsubseteq [\mathbf{1}, \mathbf{2}],\ y \sqsubseteq [\mathbf{2}, \mathbf{9}), \\
& p_1 \sqsubseteq [\langle 0, 0\rangle, \langle 1, 2\rangle],\ p_2 \sqsubseteq [\langle 1, 3\rangle, \langle 2, 9\rangle]\ \}.
\end{aligned}$$

Consider now the high level constraint plus/3 as defined in Example 5.4. This definition is overloaded and valid for the integer, real and point domains since $+_L$ *and* $-_L$ *are defined on these domains. As consequence, the constraint plus/3 may be propagated on these domains. For example,*

$$\begin{aligned}
&\{plus(r, w, t)\} \rightsquigarrow^S \{\ r \sqsubseteq [-\mathbf{89.0}, \mathbf{4.0})\ \}, \\
&\{plus(x, y, z)\} \rightsquigarrow^S \{\ z \sqsubseteq [\mathbf{3}, \mathbf{11})\ \}, \\
&\{plus(p_1, p_2, p_3)\} \rightsquigarrow^S \{\ p_3 \sqsubseteq [\langle 1, 3\rangle, \langle 3, 11\rangle]\ \}.
\end{aligned}$$

5.3 Non-Trivial Examples

In this section we show the expressivity and flexibility resulting of the cooperation of solvers in our framework.

5.3.1 Reified Constraints

Reified constraints (RCs) were defined in in Section 2.7. Basically a RC reflects the validity of a constraint into a Boolean variable. RCs are both useful and difficult to implement and normally provided as black boxes. As written in (Marriot and Stuckey, 1998, Page 284): "RCs are somewhat complex to implement since they require the solver to determine whether or not a constraint is implied by the current constraint store and whether or not its negation is implied. However because of their usefulness of RCs, some FD constraints solvers provide them". In our framework, an RC can be defined completely transparently as a high level constraint.

EXAMPLE 5.7 *In order to show how information flows between different domains, for simplicity we consider the domains Integer and Bool (although other domains may be considered). Let operators + and − be defined as in Example 4.11 on page 88 (with L = Integer). Then we define the following operators*[1] [2] *(note that* $i \in Integer^s$ *whereas* $v_1, v_2 \in Integer$*):*

$$\otimes :: Integer^s \times \overline{Bool^s} \rightarrow Integer^s \qquad \otimes' :: Integer^s \times \overline{Bool^s} \rightarrow Integer^s$$
$$i \otimes [true = i \qquad\qquad _ \otimes' [true = \bot_{Integer}]$$
$$_ \otimes [false = \top_{Integer^s} \qquad\qquad i \otimes' [false = i + \mathbf{1}]$$

$$\oslash :: Integer^s \times Bool^s \rightarrow Integer^s \qquad \oslash' :: Integer^s \times Bool^s \rightarrow Integer^s$$
$$_ \oslash true] = \top_{Integer^s} \qquad\qquad i \oslash' true] = i$$
$$i \oslash false] = i - [\mathbf{1} \qquad\qquad _ \oslash' false] = \bot_{Integer}]$$

$$\oplus :: Integer^s \times Integer^s \rightarrow Bool^s$$
$$v_1] \oplus\ v_2] = true] \qquad if\ v_1 \leq v_2,$$
$$v_1] \oplus\ v_2] = false] \qquad if\ v_1 > v_2.$$

Using the above operators, information can be made to flow between the Bool and Integer domains. Suppose that for each $b \in V_{Bool}$ *and* $x, y \in V_{Integer}$*, the high level constraint c is defined:*

$$b \equiv x \leq y \Leftrightarrow \begin{array}{ll} x & \sqsubseteq min(y)\ \overline{\otimes'}\ val(b), max(y) \otimes \overline{val(b)}, \\ y & \sqsubseteq min(x)\ \overline{\oslash'}\ \overline{val(b)}, max(x) \oslash val(b), \\ b & \sqsubseteq \overline{val(x)}\ \overline{\oplus}\ \overline{val(y)}, val(x)\ \oplus\ val(y). \end{array}$$

[1] Note that these operators do not need to be completely defined since the domains are discrete and thus "open" values such as a) are transformed to "closed" values such as $pre(a)$] by applying the equivalence rules shown in Section 4.4.4 on page 102 on both integer and Boolean domains. For instance $i \otimes (false = i \otimes [true$.

[2] For simplicity, bottom values $\bot_{Integer}) \in Integer^s$, $false) \in Bool^s$ and $(true \in \overline{Bool^s}$ are not considered since constraints using them are detected inconsistent before the evaluation of operators e.g., a constraint $x \sqsubseteq [\mathbf{1}, \bot_{Integer})$ is inconsistent.

Thus $b \equiv x \leq y$ is a reified constraint. Observe that, by imposing b to be true, $val(b)$ and $\overline{val(b)}$ are evaluated as true] and [true respectively. Then by evaluating the terms with operators, $b \equiv x \leq y$ would propagate to

$$x \sqsubseteq [\bot_{Integer}, max(y),$$
$$y \sqsubseteq min(x), \top_{Integer}]$$

which corresponds to impose the constraint $x \leq y$ as defined in (5.1) *(specialised for the Integer domain). Moreover, by imposing b to be false, $val(b)$ and $\overline{val(b)}$ evaluate as false] and [false respectively, and the constraint c would propagate to*

$$x \sqsubseteq min(y)\overline{+}[1, \top_{Integer}],$$
$$y \sqsubseteq [\bot_{Integer}, max(x) - [1$$

that corresponds exactly with the definition of the high level constraint $x > y$ in the clp(FD) system (Diaz, 1994). For instance, consider the constraint store

$$S = \{\ x \sqsubseteq [\mathbf{1}, \mathbf{7}], y \sqsubseteq [\mathbf{3}, \mathbf{8}]\ \} \in \mathcal{SS}^{\{x,y\}}.$$

Then

$$\{c\} \rightsquigarrow^{S \cup \{b=false\}} C_1$$

where

$$C_1 = \{x \sqsubseteq [\mathbf{4}, \top_{Integer}], y \sqsubseteq [\bot_{Integer}, \mathbf{6}]\}$$

and then, by store stabilisation, $S \cup C_1 \mapsto \{\ x \sqsubseteq [\mathbf{4}, \mathbf{7}], y \sqsubseteq [\mathbf{3}, \mathbf{6}]\ \}$.

Figure 5.1: The reified constraint $b \equiv x \leq y$: an example of transparent cooperation

If, however, it is known[3] *that $x \leq y$, $b \sqsubseteq [true, true]$ (i.e., $b = true$) is imposed and, as soon as $x > y$ is known to be false, $b = false$ is imposed. For instance,*

$$\{c\} \rightsquigarrow^{\{x=1, y=3, b::'Bool\}} \{b \sqsubseteq [true, true]\}.$$

[3]That is to say, x and y are ground.

Note that propagation of constraints using the indexical term $val(z)$ (or its mirror) for $z \in \{x, y, b\}$, is delayed until z is ground to any value m (and then the constraint $z \sqsubseteq [\mathbf{m}, \mathbf{m}]$ is imposed).

Figure 5.1 shows graphically how the information can flow between the domains of the variables x, y and b.

5.3.2 Propagation on Direct Combinations

Consider the following example taken from (Pachet and Roy, 1995) in the domain of planar geometry. This is a simplified form of a problem occurring in the field of robot mobility to avoid a crash between a rectangular robot with another rectangular object (e.g., a table, a corner or other robot).

Problem Statement. Find all pairs of non-trivial quadrilaterals satisfying the following set of constraints:

1. C_1: All vertices have integer coordinates in $\{1..n\}$.
2. C_2: All quadrilaterals are straight rectangles.
3. C_3 : The two rectangles do not intersect.

Representation of the problem. Given the plane $(0, 0) \times (n, n)$ a rectangle r is identified by its lower left corner and its upper right corner (for simplicity, the set of constraints $\{x \leq y, y \sqsubseteq [l, l]\}$ where $x, y \in V_L$ and $l \in L$ will just be denoted as $x \leq l$. And analogously for $l \leq x$).

In the following we show how it is possible to solve this problem on different domains by using the generic constraint $\leq /2$ defined in Example 5.3.

Designing the problem in the integer domain.

Let $integer^+$ be the (lifted) lattice of the positive integers plus value 0 where $\bot_{integer^+} = 0$ and $\top_{integer^+}$ is a ficticial bound. A first formulation consists of identifying a rectangle by the coordinates of its corners and consider them as atomic values in the $integer^+$ domain. Constraints are stated entirely in terms of the $integer^+$ domain.

Let (p_{1x}, p_{1y}),(p_{2x}, p_{2y}) and (p_{3x}, p_{3y}),(p_{4x}, p_{4y}) be the coordinates of the two corners identifying a rectangle r_1 (e.g., a robot) and r_2 (e.g., a table) respectively (see diagram (a) in Figure 5.2). Constraint C_1 can be stated as follows:

$$\begin{aligned}
&(1 \leq p_{1x}) \wedge (p_{1x} \leq n) \wedge (1 \leq p_{1y}) \wedge (p_{1y} \leq n) \wedge \\
&(1 \leq p_{2x}) \wedge (p_{2x} \leq n) \wedge (1 \leq p_{2y}) \wedge (p_{2y} \leq n) \wedge \\
&(1 \leq p_{3x}) \wedge (p_{3x} \leq n) \wedge (1 \leq p_{3y}) \wedge (p_{3y} \leq n) \wedge \\
&(1 \leq p_{4x}) \wedge (p_{4x} \leq n) \wedge (1 \leq p_{4y}) \wedge (p_{4y} \leq n).
\end{aligned}$$

C_2 as follows:

$$(p_{1x} < p_{2x}) \wedge (p_{1y} < p_{2y}) \wedge (p_{3x} < p_{4x}) \wedge (p_{3y} < p_{4y})$$

where constraint $</2$ is just defined as $x < y \Leftrightarrow x \leq y \wedge x \neq y$. And C_3 as:

$$(p_{2x} < p_{3x}) \vee (p_{2y} < p_{3y})$$

Designing the problem in the *Point* domain.

As already shown, our lattice-based framework allows for new computation domains to be constructed from previously defined domains. The *Point* domain can be constructed from the direct product $\langle integer^+, integer^+ \rangle$. Note that $\bot_{Point} = \langle 0, 0 \rangle$ and $\top_{Point} = \langle \top_{integer^+}, \top_{integer^+} \rangle$.

Figure 5.2: The problem of non-intersecting rectangles

Then, r_1 and r_2 can be identified by the points p_1,p_2 and p_3,p_4 respectively (see diagram (b) in Figure 5.2) and constraints C_1, C_2 and C_3 can be stated directly in the *Point* domain. Constraint C_1 is stated as follows:

$$(\langle 1, 1 \rangle \leq p_1) \wedge (p_1 \leq \langle n, n \rangle) \wedge (\langle 1, 1 \rangle \leq p_2) \wedge (p_2 \leq \langle n, n \rangle) \wedge$$
$$(\langle 1, 1 \rangle \leq p_3) \wedge (p_3 \leq \langle n, n \rangle) \wedge (\langle 1, 1 \rangle \leq p_4) \wedge (p_4 \leq \langle n, n \rangle),$$

constraint C_2 as $(p_1 < p_2) \wedge (p_3 < p_4)$ and constraint C_3 should be stated as $p_3 \not\leq p_2$. However, it is not clear how to express this last constraint since the meaning is not exactly $p_2 < p_3$ (note that there are values in the point domain such that $p_2 \not\sim p_3$). In the following we propose a solution based on the concept of solver cooperation.

Designing the problem by solver cooperation. As it was shown above, constraint C_3 can easily be defined in the $integer^+$ domain. A solution can be formulated via combining this definition with the adequate definition of an operator to allow the flow of information between the $integer^+$ domain and the *Point* domain.

For example, let $L, L_1 \in \mathcal{L}$ where $L_1 = \langle L, L\rangle$ and $\diamond$ defined generically as:

$$\diamond :: L^s \times L^s \to L_1^s$$
$$a\}_1 \diamond b\}_2 = \langle a, b\rangle min_B(\}_1, \}_2).$$

Thus, if $L = Integer^+$, we have **3)** $\diamond$ **2]** $= \langle 3, 2\rangle$**)** whereas if $L = Point$, $\langle 1, 3\rangle] \diamond \langle 4, 4\rangle] = \langle\langle 1, 3\rangle, \langle 4, 4\rangle\rangle]$. Observe that the operator $\diamond$ generates a one-way channel that allows information to propagate from the domain L to the domain $\langle L, L\rangle$.

Now C_1 and C_2 are defined as in the formulation for the *Point* domain shown above whereas C_3 is defined as in the formulation for the $integer^+$ domain also shown above. To move information resulting from constraint propagation from one domain to the other one, we can define the following constraints

$$\begin{aligned} p_2 &\sqsubseteq min(p_{2x}) \mathbin{\overline{\diamond}} min(p_{2y}), max(p_{2x}) \diamond max(p_{2y}), \\ p_3 &\sqsubseteq min(p_{3x}) \mathbin{\overline{\diamond}} min(p_{3y}), max(p_{3x}) \diamond max(p_{3y}). \end{aligned} \tag{5.2}$$

Then any change in the coordinates $p_{2x}, p_{2y}, p_{3x}, p_{3x}$ is directly propagated, by evaluation of the indexical terms, to points p_2 and p_3.

Let now *Rect* the rectangle domain as defined in Section 4.6.3 where *Point* is as defined above (i.e., $Rect = \langle Point, Point\rangle$). Note that $\bot_{Rect} = \langle\langle 0, 0\rangle, \langle 0, 0\rangle\rangle$ and $\top_{Rect} = \langle \top_{Point}, \top_{Point}\rangle$. Then information can be propagated from the domain *Point* to the domain *Rect* by using the same operator $\diamond$ defined above. Let $r_1, r_2 \in V_{Rect}$ and $p_1, p_2, p_3, p_4 \in V_{Point}$. Then values in the *Point* domain can be directly propagated to the *Rect* domain via the following constraints:

$$r_1 \sqsubseteq min(p_1) \mathbin{\overline{\diamond}} min(p_2), max(p_1) \diamond max(p_2)$$
$$r_2 \sqsubseteq min(p_3) \mathbin{\overline{\diamond}} min(p_4), max(p_3) \diamond max(p_4)$$

where $r_1, r_2 \in V_{Rect}$ and $p_1, p_2, p_3, p_4 \in V_{Point}$.

5.3.3 A More Motivating Example on Linear Combinations

In this section we show that our extension is useful not only on the direct or lexicographic combination of domains but also on more interesting combinations such as the linear sum of domains.

In current CLP systems, most of the arithmetical operations executed on a particular domain return a value belonging to such a domain. For example, an expression such as $a + b$, where a, b are reals, returns a real value. However, in practice there are well-known domains such as the binary and hexadecimal domains in which this is not necessarily true (see Figure 5.3). For example, in the hexadecimal domain, 'F'+'F' does not belong to the hexadecimal domain but to the lexicographic product $(hexadecimal, hexadecimal)$. This means that the standard definition of the operator + is not valid.

A naive solution would extend the operation + with an extra *carry* argument. For example, F+F=(1,E). In the following we propose a more expressive and elegant solution defined in our setting and based the concept of solver collaboration.

Figure 5.3: Linear combinations of the sum of domains

Let *0toF* be the lattice of hexadecimal digits as defined in Section 4.6.3 on page 118 (i.e., $0toF = 0to9 \uplus AtoF$) and $0toF^2$ the lexicographic product domain $(0toF, 0toF)$. Now we overload the operators + and − shown in Example 4.11 on page 88 by defining

$$+ :: 0toF^s \times 0toF^s \to 0toF^{2s} \qquad - :: 0toF^{2s} \times \overline{0toF^s} \to 0toF^s$$

$$h_1\}_1 + h_2\}_2 = h_1 +_{0toF^2} h_2 \ \ \}_1 +_B \}_1 \qquad (h_1, h_2)\}_1 - \{_2 h_3 = (h_1, h_2) -_{0toF} \hat{h_3} \ \ \}_1 +_B \}_2$$

where $+_B$ is defined as in Example 4.11 and $+_{0toF^2}$ and $-_{0toF}$ are defined as usual in the hexadecimal domain i.e.,

$$+_{0toF^2} :: 0toF \times 0toF \to 0toF^2 \qquad -_{0toF} :: 0toF^2 \times \widehat{0toF} \to L$$

$$\begin{array}{ll}
\text{`0'} +_{0toF^2} \text{`0'} = (\text{`0'},\text{`0'}), & (\text{`0'},\text{`0'}) -_{0toF} \hat{\text{`0'}} = \text{`0'}, \\
\ldots & \ldots \\
\text{`1'} +_{0toF^2} \text{`F'} = (\text{`1'},\text{`0'}), & (\text{`1'},\text{`0'}) -_{0toF} \hat{\text{`F'}} = \text{`1'}, \\
\ldots & (\text{`1'},\text{`0'}) -_{0toF} \hat{\text{`1'}} = \text{`F'}, \\
\ldots & \ldots, \\
\text{`F'} +_{0toF^2} \text{`F'} = (\text{`1'},\text{`E'}), & (\text{`1'},\text{`E'}) -_{0toF} \hat{\text{`F'}} = \text{`F'}, \\
\ldots & \ldots
\end{array}$$

Then,

$$\text{`1'}]+[\text{`F'}] = (\text{`1'},\text{`0'})] \text{ and } (\text{`1'},\text{`0'})] - [\text{`1'} = \text{`F'}].$$

As consequence, the high level constraint $plus/3$ defined in Example 5.4 can be used on the hexadecimal if we consider $x, y \in V_{0toF}$ and $z \in V_{0toF^2}$.

For instance, let $h_1, h_2 \in V_{0toF}$ and $h_3 \in V_{0toF^2}$, and consider the following constraint store:

$$S = \{\ h_1 \sqsubseteq [\text{`0'},\text{`F'}], h_2 \sqsubseteq (\text{`A'},\text{`F'}]\ \}.$$

Then

$$\{\ plus(h_1, h_2, h_3)\ \} \rightsquigarrow^S \{\ h_3 \sqsubseteq [(\text{`0'},\text{`A'}),(\text{`1'},\text{`E'})]\ \}.$$

Additional constraints can also be applied. Consider now the generic high level constraint $\leq/2$ as defined in Example 5.3. Then a constraint such as $h_3 \leq (\text{`1'},\text{`E'})$ restricts h_3 to the range[4][5] $[(\text{`0'},\text{`0'}),(\text{`1'},\text{`E'})]$. Moreover, the flexibility of our framework

[4]Observe that the maximum of the sum of two single hexadecimals is (‘1’,‘E’).

[5]Remember that $\bot_{L_1} = (\text{`0'},\text{`0'})$.

allows alternative formulations. For example, let $\oslash/2$ be a generic operator defined as the operator $\diamond$ in Section 5.3.2 but where $L_1 = (L, L)$, that is to say,

$$\oslash :: L^s \times L^s \to {L_1}^s$$
$$a\}_1 \oslash b\}_2 = (a, b)\ min_B(\}_1, \}_2),$$

and let $h_4 \in V_{0toF}$. Then

$$\{\ h_3 \sqsubseteq min(h_4)\ \overline{\oslash}\ [\text{'0'}, max(h_4) \oslash \text{'E'}]\ \} \rightsquigarrow^{\{\ h_4 \sqsubseteq [\text{'0'},\text{'1'}]\ \}} \{\ h_3 \sqsubseteq [(\text{'0'},\text{'0'}),(\text{'1'},\text{'E'})]\ \}.$$

This example provides an idea of the flexibility and expressivity of the framework.

5.4 Even More Expressivity!

So far, we have shown that our setting allows constraint propagation from basic domains to combined domains and vice-versa. In this section we show by example that it also enables information to flow between non-related domains.

EXAMPLE 5.8 *Let $\Re_0$ be the set of non-negative reals and $Integer_0$ the set of non-negative integers. Consider the following conditional sentence*[6]*:*

$$\textbf{if } b \textbf{ then } i \leftarrow trunc(r * r) \textbf{ else } i \leftarrow trunc(r) \textbf{endif}$$

where $b \in Bool$, $r \in \Re_0$ and $i \in Integer_0$. The meaning of this conditional sentence can be captured in the interval setting by the following interval constraint:

$$c_1 \ = \ i \sqsubseteq min(b)\overline{\rhd}min(r), max(b) \rhd max(r)$$

where $r \in V_{\Re_0}$, $b \in V_{Bool}$, $i \in V_{Integer_0}$ and $\rhd$ is a constraint operator for ${Integer_0}^s$ declared and defined as follows

$$\rhd :: {Bool}^s \times {\Re_0}^s \to {Integer_0}^s$$
$$true] \rhd r\} = trunc_{Integer}(r * r)]$$
$$false] \rhd r\} = trunc_{Integer}(r)].$$

where $trunc_{Integer}(a)$ returns the integer part of a for any $a \in \Re_0$. Observe also that, as the Boolean domain is discrete, we only consider "closed" values since "open" values can be reduced to "closed" ones by applying the equivalence rules shown in Section 4.4.4 on page 102.

Constraint propagation is executed as usual. For example, let

$$S = \{\ r \sqsubseteq [\mathbf{2.3}, \mathbf{8.9}), b \sqsubseteq [false, true]\ \}.$$

Then[7]

$$\{c_1\} \rightsquigarrow^S \{\ i \sqsubseteq [\mathbf{2}, \mathbf{79}]\ \}.$$

[6] The expression $x \leftarrow e$ means "assign the value returned by the expression e to the variable x".

[7] Note that $i \sqsubseteq [\mathbf{2}, \mathbf{79}]$ comes from $i \sqsubseteq [false\overline{\rhd}[2.3, true] \rhd 8.9)$.

More propagation is obtained by constraint stabilisation. For instance,

$$S \cup \{b \sqsubseteq [\mathit{false}, \mathit{false}]\} \mapsto S_1$$

and $S_1 = \{r \sqsubseteq [\mathbf{2.3}, \mathbf{8.9}), b \sqsubseteq [\mathit{false}, \mathit{false}]\ \}$. *Then*[8] $\{c_1\} \rightsquigarrow^{S_1} \{\ i \sqsubseteq [\mathbf{2}, \mathbf{8}]\ \}$.

5.5 An Application (in Biomedicine) with Uncertainty

To demonstrate the practical nature of our framework, we describe here an application where there is uncertainty in the knowledge base so that an explicit handling of vagueness and uncertainty is required. This application involves high level constraints, solver cooperation and domain combination. The application is taken from a real diagnosis problem in biomedicine where test results contain a margin of error depending on the precision of the measuring instrument.

5.5.1 Representing a Margin of Error

We first define a function that corrects for an error measured in a numerical domain e.g., $\Re$. For each $i \in \mathit{Integer}$ and $n, r \in \Re$, we define the constraint operator $\gg$ when $L_1 = \Re$ and, for $L = \Re$ and for $L = \mathit{Integer}$, as:

$$\begin{aligned} &\gg :: L^s \times L_1^s \to L^s \\ &\mathbf{n}\}_1 \gg \mathbf{r}\}_2 = \mathbf{n}\}_1 + \mathbf{r}\}_2 \\ &\mathbf{i}\}_1 \gg \mathbf{r}\}_2 = \mathbf{i}\}_1 + \mathit{round}_{\Re}(r)] \end{aligned}$$

where operator $+$ is defined (for $\Re^s$ and $\mathit{Integer}^s$) in Example 4.11 on page 88 and $\mathit{round}_{\Re}(r)$ returns the rounded integer of r (e.g., $\mathit{round}_{\Re}(-2.6) = -3$ and $\mathit{round}_{\Re}(2.6) = 3$). For instance,

$$\begin{aligned} &3.2) \gg 2.6) \ = \ 3.2) + 2.6) \ = \ 5.8), \\ &3) \gg 2.6) \ = \ 3) + 3] \ = \ 6). \end{aligned}$$

If $\alpha \in V_{L_1}$, we define a symbolic high level constraint for an *approximate equality*:

$$x \approx^{\alpha} y \Leftrightarrow \quad \begin{array}{l} x \sqsubseteq \mathit{min}(y) \overline{\gg} \mathit{min}(\alpha), \mathit{max}(y) \gg \mathit{max}(\alpha), \\ y \sqsubseteq \mathit{min}(x) \overline{\gg} \mathit{min}(\alpha), \mathit{max}(x) \gg \mathit{max}(\alpha). \end{array}$$

where $L, L_1 \in \mathcal{L}$ and $x, y \in V_L$. α is a variable (the *cut*) of L_1 that represents the degree of (in-)equality for x and y. If $x \approx^{\alpha} y$ is true, then we say that x and y are equal via α. Note that this constraint is overloaded since it can be used for any domains L, L_1 in which $\gg$ is defined.

The meaning of a constraint $x \approx^{\alpha} y$ is that x and y are considered equal if we take into account a possible error bound α (where α is associated to an error interval). For instance, a test result such as systolic blood pressure has an error margin due to the

[8] Note that $i \sqsubseteq [\mathbf{2}, \mathbf{8}]$ comes from $i \sqsubseteq [\mathit{false} \overline{\rhd} [\mathit{2.3}, \mathit{false}] \rhd \mathit{8.9})$.

measurement's degree of accuracy. Suppose this margin is known to be ± 2.6 so that we have the constraint $\alpha \sqsubseteq$ **[-2.6,2.6]**. Consider the constraint c defined as

$$value \approx^{\alpha} reading$$

where $reading, value \in V_{Integer}$ and the store

$$S = \{\ \alpha \sqsubseteq \textbf{[-2.6,2.6]},\ value ::' Integer\ \}.$$

Suppose now that a patient has an approximate measurement of 117 (i.e.,[9] $reading = 117$). Then, as usual in constraint propagation,

$$c \rightsquigarrow^{S \cup \{reading=117\}} \left\{ \begin{array}{c} value \sqsubseteq min(reading) \overline{\gg} min(\alpha), max(reading) \gg max(\alpha), \\ reading \sqsubseteq min(value) \overline{\gg} min(\alpha), max(value) \gg max(\alpha) \end{array} \right\}$$

that, by evaluation of the indexical terms, is equivalent to

$$c \rightsquigarrow^{S \cup \{reading=117\}} \left\{ \begin{array}{c} value \sqsubseteq \textbf{[117} \overline{\gg} \textbf{[-2.6, 117]} \gg \textbf{2.6]}, \\ reading \sqsubseteq [\bot_{Integer} \overline{\gg} \textbf{[-2.6}, \top_{Integer}] \gg \textbf{2.6]} \end{array} \right\}$$

and, thus by evaluating the operators,

$$c \rightsquigarrow^{S \cup \{reading=117\}} \left\{ \begin{array}{c} value \sqsubseteq \textbf{[114,120]}, \\ reading \sqsubseteq [\bot_{Integer}, \top_{Integer}] \end{array} \right\}$$

what is interpreted as meaning that the true reading value is between 114 and 120.

5.5.2 The Problem of Diagnosing the Heart Functionality

The analysis of medical data often uses a combination of different kinds of domains for diagnosing illness. Consider the three-category problem of differentiating among normal heart function, myocardial infarction and angina pectoris (Hudson and Cohen, 2000). The heart condition is represented by the combined domain

$$\begin{array}{rl} heart = \langle Category, \Re, Integer \rangle & \text{where} \\ Category = \{none, low, high\} & \text{with ordering } none \prec low \prec high. \end{array}$$

For each component of the *heart* domain, we define a variable to denote its observed value:

(1) $pvc \in Category$ denotes the rhythm of the postventricular contractions;

(2) $pr \in \Re$ denotes the measured value of the pulse rate and

(3) $wbc \in Integer$ denotes the white blood cell count.

Table 5.1: Ranges for heart function

State	pvc's	pr	wbc
Normal	none-none	[40.0,100.0]	[4800,13500]
Angina pectoris	none-none	≥ 105.0	[4800,13500]
Myocardial Infarction	low-high	≥ 105.0	> 13500

Table 5.1 summarises the three category problem for the heart function. The values of *pr* and *wbc* have a certain degree of imprecision (known to be ± 1.0 and ± 45.7 respectively) and we denote the amount of uncertainty for *wbc* and *pr* by α_{wbc} and α_{pr} respectively. To allow for uncertainty in the value for *heart*, we use the constraint $\approx^{\alpha}$ as defined in Section 5.5.1, together with the following overloaded definition for the operator $\gg$ (with $L = heart$ and $L_1 = \Re^2$):

$$\gg :: heart^s \times (\Re^2)^s \to heart^s$$
$$\langle pvc, pr, wbc\rangle\}_1 \gg \langle \alpha_{pr}, \alpha_{wbc}\rangle\}_2 = \langle pvc, pr + \alpha_{pr}, wbc + round_{\Re}(\alpha_{wbc})\rangle\ ].$$

Suppose now that *STATE* is the computation domain:

$$error_1 \prec normal \prec error_2 \prec angina \prec error_3 \prec infarction \prec error_4.$$

and also that we have the operators[10]:

$$\begin{array}{ll}
\Diamond :: STATE^s \to heart^s & \Diamond :: STATE^s \to heart^s \\
\Diamond'\ error_1] = \bot_{heart^s}; & \Diamond\ error_1] = \bot_{heart^s}; \\
\Diamond'\ normal] = \langle none, 40.0, 4800\rangle]; & \Diamond\ normal] = \langle none, 100.0, 13500\rangle]; \\
\Diamond'\ error_2] = \langle none, 100.0, 4800\rangle); & \Diamond\ error_2] = \langle none, 105.0, 13500\rangle]; \\
\Diamond'\ angina] = \langle none, 105.0, 4800\rangle]; & \Diamond\ angina] = \langle none, \top_{\Re}, 13500\rangle]; \\
\Diamond'\ error_3] = \langle none, 105.0, 13500\rangle); & \Diamond\ error_3] = \langle low, \top_{\Re}, 13500\rangle]; \\
\Diamond'\ infarction] = \langle low, 105.0, 13501\rangle]; & \Diamond\ infarction] = \top_{heart^s}; \\
\Diamond'\ error_4] = \top_{heart^s}; & \Diamond\ error_4] = \top_{heart^s}.
\end{array}$$

[9] What is the same as telling *reading* $\sqsubseteq [117, 117]$.

[10] Note that operators $\Diamond$ and $\Diamond'$ do not need to be complete functions by the equivalence rules shown in Section 4.4.4 on page 102, e.g., $\Diamond angina) = \Diamond error_2]$.

$$
\begin{aligned}
\bowtie :: & \; heart^s \to STATE^s \\
\bowtie reading = error_1] & \text{ if } reading \succeq_{heart^s} \bot_{heart^s} \\
& \text{and } (reading \preceq_{heart^s} \langle high, \top_{\Re}, 4799\rangle] \\
& \quad \text{or } \; reading \preceq_{heart^s} \langle high, pr, \top_{Integer}\rangle] \text{ and } pr < 40.0 \text{ }); \\
\bowtie reading = normal] & \text{ if } reading \succeq_{heart^s} \langle none, 40.0, 4800\rangle] \\
& \text{and } reading \preceq_{heart^s} \langle none, 100.0, 13500\rangle]; \\
\bowtie reading = error_2] & \text{ if } reading \succeq_{heart^s} \langle none, pr_1, 4800\rangle] \text{ and } pr_1 > 100.0 \\
& \text{and } reading \preceq_{heart^s} \langle none, pr_2, 13500\rangle] \text{ and } pr_2 < 105.0; \\
\bowtie reading = angina] & \text{ if } reading \succeq_{heart^s} \langle none, 105.0, 4800\rangle] \\
& \text{and } reading \preceq_{heart^s} \langle none, \top_{\Re}, 13500\rangle]; \\
\bowtie reading = error_3] & \text{ if } reading \succeq_{heart^s} \langle none, 40.0, 13501\rangle] \\
& \text{and } reading \preceq_{heart^s} \langle high, pr, \top_{Integer}\rangle] \text{ and } pr < 105.0; \\
\bowtie reading = infarction] & \text{ if } reading \succeq_{heart^s} \langle low, 105.0, 13501\rangle]; \\
\bowtie reading = error_4] & \text{ otherwise.}
\end{aligned}
$$

Note that values $error_i$ for $1 \leq i \leq 4$ are added to the $STATE$ domain to capture atypical readings[11] and to maintain the monotonicity of $\bowtie$.

Let $reading, true\_reading \in V_{heart}$ and $state \in V_{STATE}$. The state of the heart can be diagnosed by the correct definition of a high level constraint as follows

$$
\begin{aligned}
diagnostic(reading, state, \alpha) \Leftrightarrow reading \approx^{\alpha} true\_reading, & \\
state \sqsubseteq \bowtie(min(true\_reading)), \bowtie (max(true\_reading)), & \\
true\_reading \sqsubseteq \overline{\Diamond'}(min(state)), \Diamond(max(state)). &
\end{aligned}
$$

This constraint subsumes the heart evolution and relates the domains $STATE$, $heart$ and $\Re^2$ by opening two-way channels in which information can flow from one domain to each another, that is to say, given an observed value (i.e., an interval) in the variable $reading$, this is propagated to the variable $state$ by taking into consideration the possible measurement errors associated to the variable α; also, a state value can be propagated from the state variable to the reading variable.

For instance, suppose that $reading = \langle none, 85.6, 10000\rangle$ and consider the values for α_{pr} and α_{wbc} provided above. To diagnose the state of the heart we have then to solve the set of constraints $C = \{c_1, c_2, c_3\}$ where

$$
\begin{aligned}
c_1 &\equiv \alpha \sqsubseteq [\langle -1.0, -45.7\rangle, \langle 1.0, 45.7\rangle], \\
c_2 &\equiv reading \sqsubseteq [\langle none, 85.6, 10000\rangle, \langle none, 85.6, 10000\rangle], \\
c_3 &\equiv diagnostic(reading, state, \alpha).
\end{aligned}
$$

[11] It could happen that, for some persons, the observed values are not in the ranges of Table 5.1. This is an atypical situation to be studied by the doctor and then we return an error value.

Table 5.2: Solving sequence in the problem of heart functionality diagnosis

Initialisation	$S_0 = S = \{\ \alpha \sqsubseteq [\bot_{\Re^2}, \top_{\Re^2}],\ \mathit{reading} \sqsubseteq [\bot_{\mathit{heart}}, \top_{\mathit{heart}}],$ $\mathit{true\_reading} \sqsubseteq [\bot_{\mathit{heart}}, \top_{\mathit{heart}}],\ \mathit{state} \sqsubseteq [\mathit{error}_1, \mathit{error}_4]\ \}$
By line (1)	$C = \{c_1, c_2, c_3\} \cup S_0$
1st iteration	
$C \rightsquigarrow^S C'$	$C' = \{c_1, c_2\} \cup S_0$
$S' \cup C' \mapsto S$	$S = \{\ c_1, c_2,\ \mathit{true\_reading} \sqsubseteq [\bot_{\mathit{heart}}, \top_{\mathit{heart}}], \mathit{state} \sqsubseteq [\mathit{error}_1, \mathit{error}_4]\ \}$
2nd iteration	
$C \rightsquigarrow^S C'$	$C' = \{c_1, c_2, \mathit{true\_reading} \sqsubseteq [\langle \mathit{none}, 84.6, 9954\rangle, \langle \mathit{none}, 86.6, 10046\rangle]\ \} \cup S_0$
$S' \cup C' \mapsto S$	$S = \{c_1, c_2, \mathit{state} \sqsubseteq [\mathit{error}_1, \mathit{error}_4],$ $\mathit{true\_reading} \sqsubseteq [\langle \mathit{none}, 84.6, 9954\rangle, \langle \mathit{none}, 86.6, 10046\rangle]\ \}$
3rd iteration	
$C \rightsquigarrow^S C'$	$C' = \{c_1, c_2, \mathit{reading} \sqsubseteq [\langle \mathit{none}, 83.6, 9900\rangle, \langle \mathit{none}, 87.6, 10100\rangle],$ $\mathit{true\_reading} \sqsubseteq [\langle \mathit{none}, 84.6, 9954\rangle, \langle \mathit{none}, 86.6, 10046\rangle],$ $\mathit{state} \sqsubseteq [\mathit{normal}, \mathit{normal}]\ \} \cup S_0$
$S' \cup C' \mapsto S$	$S = \{c_1, c_2, \mathit{state} = \mathit{normal},$ $\mathit{true\_reading} \sqsubseteq [\langle \mathit{none}, 84.6, 9954\rangle, \langle \mathit{none}, 86.6, 10046\rangle]\ \}$
4th iteration	
$C \rightsquigarrow^S C'$	$C' = \{c_1, c_2, \mathit{state} = \mathit{normal},$ $\mathit{reading} \sqsubseteq [\langle \mathit{none}, 83.6, 9900\rangle, \langle \mathit{none}, 87.6, 10100\rangle],$ $\mathit{true\_reading} \sqsubseteq [\langle \mathit{none}, 84.6, 9954\rangle, \langle \mathit{none}, 86.6, 10046\rangle],$ $\mathit{true\_reading} \sqsubseteq [\langle \mathit{none}, 40.0, 4800\rangle, \langle \mathit{none}, 100.0, 13500\rangle]\ \} \cup S_0$
$S' \cup C' \mapsto S$	$S = \{c_1, c_2, \mathit{state} = \mathit{normal}$ $\mathit{true\_reading} \sqsubseteq [\langle \mathit{none}, 84.6, 9954\rangle, \langle \mathit{none}, 86.6, 10046\rangle]\ \}$
END	The procedure terminates since $S = S'$

By executing the operational schema $\mathit{solve}(C, S)$, as defined in Section 4.5.1, with S initialised to the top element of $\mathcal{SS}^{\{\mathit{state}, \mathit{reading}, \mathit{true\_reading}, \alpha\}}$, the information is propagated from the *heart* to the *STATE* domain and viceversa and the following solution is found after four iterations of the *repeat* loop

$$\begin{aligned} S = \{&c_1, c_2,\\ &\mathit{state} = \mathit{normal},\\ &\mathit{true\_reading} \sqsubseteq [\langle \mathit{none}, 84.6, 9954\rangle, \langle \mathit{none}, 86.6, 10046\rangle]\ \}. \end{aligned}$$

Table 5.2 shows the contents of the stores C' and S in each of the iterations during the execution of the procedure $\mathit{solve}(C, S)$[12] where S is initialised to the top element of the lattice $\mathcal{SS}^{\{\alpha, \mathit{reading}, \mathit{true\_reading}, \mathit{state}\}}$. The conclusion is that the heart is found

[12]Note that the schema $\mathit{solve}_\varepsilon(C, S)$ is not necessary in this example since termination is reached even if $\varepsilon = 0.0$.

to be functioning normally. Observe that during the solving sequence the constraint $reading \approx^{\alpha} true\_reading$ is propagated to

$$true\_reading \sqsubseteq [\langle none, 84.6, 9954\rangle, \langle none, 86.6, 10046\rangle].$$

what means that the error of measure instruments was having into account.

5.6 Related Work

Solver cooperation is an important issue for the constraint applications. The interaction between solvers makes it easier to express compound problems and good communication can help the efficiency of the systems. Existing cooperative solvers are very diverse and range from domain combinations to a mix of distinct techniques for solving constraints over the same domain. Moreover, the cooperating solvers may be very different in nature: some of them can perform complete constraint solving whereas others can execute simple forms of propagation. In this section we discuss part of the relevant constraint literature and compare their proposals with our own. Note that the issues of communication and cooperation are relevant to many other aspects of computation: here, we only consider proposals closely related to our own.

There are a number of constraint systems that provide support for the interaction between solvers defined over built-in and predefined domains. In these systems, the concept of *solver* is viewed as a box that transforms the original set of constraints to an equivalent reduced set. In this category, we have, for example, the following systems:

- CLP(BNR) (Benhamou and Older, 1997), Prolog III (Colmerauer, 1990) and Prolog IV (N'Dong, 1997) allow solver cooperation, mainly limited to Booleans, reals and naturals (and also to lists and trees). Observe that our framework easily emulates this kind of cooperation since all these domains can be naturally defined as lattices.

- The language NCL (Zhou, 2000a) provides an integrated constraint framework that strongly combines Boolean logic, integer constraints and set reasoning. In (Zhou, 2000a), the integration of new constraint domains such as the reals is described as future work.

  We have shown that our framework makes possible not only the communication between the Boolean, integer, reals and set domains but also the combination of them by means of lattice combinators.

The two main problems with most of these cooperative existing systems are: (1) the cooperation is restricted to a limited set of computation domains supported by the system and (2) the solvers are usually black boxes so that it is difficult (sometimes impossible) to extend the original set of constraints. As shown throughout this document, these are no such problems in our framework.

Another kind of cooperation consists in providing special built-in constructs with the functionality to propagate information from one domain to another. This is the

case with the reified constraints described in Section 2.7 that enable arithmetic values to be propagated to the Boolean domain and vice-versa. In this chapter we have shown that reified constraints can be defined in our framework at the user level.

In the same class of cooperation we can cite *Conjunto* (Gervet, 1997) which is a constraint language for propagating interval constraints defined over finite sets of integers. This language also provides a set of constraints called *graduated constraints* which map sets onto arithmetic terms, allowing thus a one-way cooperative channel from the set domain to the integer domain. Graduated constraints can be used in a number of applications as for instances to handle optimisation problems by applying a cost function to the quantifiable terms (i.e., arithmetic terms which are associated to set terms). This form of interaction is easily emulated in our framework by defining constraint operators from the set to the integer domain. Moreover, our framework allows the concept of graduated constraints to be extended since communication may also be enabled from integers to sets by means of suitable definitions of constraint operators.

Recently, another approach for solver cooperation has appeared. This approach requires *interoperatibility* which means that the system has the ability to communicate and use independently written software components, thus making independent systems cooperate. Usually to add a solver requires the addition of an interface between the new solver and the existing ones. The format of the inputs and outputs of the interface (and thus of the solvers) must be specified precisely.

For instance, (Goualard, 2001) proposes a C++ constraint solving library called aLiX for communicating between different solvers, possibly written in different languages. Two of the main aims of aLiX are to permit the transparent communication of solvers and assure *type safety*, that is to say, the capacity to prevent *a priori* the connection of a solver that does not understand the input format of the interface with another solver. The current version of aLiX is not mature yet although its interoperative approach offers interesting possibilities. One of the main shortcomings of the current aLiX version is that a component for solving continuous constraints is not integrated into the system yet and thus real constraints cannot be processed (this is claimed to be one of the main priorities for future development work).

In the same spirit, for the real domain, many constraint systems provide both a linear and a non-linear solver. As the linear solver is the most efficient of the two, this should be used whenever the constraints are linear. Thus there is a need for communication between the solvers. As an example, (Monfroy et al., 1995) describes a client/server architecture to enable communication between the component solvers. This consists of both managers of the system and the solvers that must be defined on the same computational domain (as real numbers for example) but with different classes of admissible constraints (i.e., linear and non-linear constraints). The CLP system *CoSAc* is an implementation of their system. A built-in platform permits the integration and connection of the components. The exchange of information is managed by means of pipes and the data that is exchanged is a character string. One of the main drawbacks of this system is the lack of type safety. Also, the cooperation happens at a fixed level that avoids the communication of solvers in a transparent way since the solvers cannot

obtain additional information from the structure of the internal constraint store. The use of this information may lead to speeding up the solving process.

Note that, for our framework, the generic solver used for all the domains can only handle linear constraints. The question of how our framework may be extended to handle simple non-linear constraints such as $ax^2 + b = y$ in the real domain is ongoing and future work. A solution may be in the combination of our propagation mechanism with other interval technique for solving non-linear systems. This is an issue of further work.

As CoSAc does not permit solver combination, Monfroy designed a domain independent environment for solver collaboration[13] (i.e., solver cooperation plus solver combination) and constructed the system **BALI** that facilitates the integration of heterogeneous solvers as well as the specification of solver cooperation (Monfroy, 1996). In his thesis, Monfroy also designed **SoleX**, a method for extending constraint solvers with new function symbols. Unfortunately, although it is commented in (Monfroy, 1996) (page 195), **SoleX** and **BALI** were not integrated. Observe that such an integration could lead to a framework including either solver collaboration and/or solver extension.

There are several papers that focus on the combination of distinct interval solving methods (basically on the real domain). For instance, (Benhamou, 1996) describes a unified framework for heterogeneous constraint solving. The main idea is to represent the solvers as constraint narrowing operators (CNO), that are closure operators, and use a generalised notion of arc consistency. The necessary conditions of the CNOs to assure the main properties of the principal algorithm are provided and it is shown how the solvers can communicate, share common variables and send and receive information to each other. In spite of the fact that it is not very clear the similarities of this work with respect to our proposal, we observe main differences. For instance: (1) in the system described in (Benhamou, 1996) the cooperation comes from the combination of different algorithms (possibly defined over distinct structures) whereas in our framework the cooperation does not depend on the propagation algorithm since this is common to all the structures (i.e., domains) supported by the system; (2) as we also do, (Benhamou, 1996) gives a fixed point semantics to describe the cooperation process. However, the termination of the central algorithm proposed in (Benhamou, 1996) relies on the finiteness of the computation domain[14]. Thus, termination cannot be guaranteed in the case of non-finite domains as it can be the case of the domain of "sets of reals". Moreover, real interval constraints are viewed as a generalisation of the finite intervals with a floating point precision for the real domain. In our framework, reals can be considered with no restriction.

Also (Benhamou et al., 1999) proposes, to solve real constraints, the combination of hull consistency and box consistency (see Section 2.6.5 on page 38) with the objective to reduce the computation time of using box consistency alone. This idea was reflected

[13] We believe that the notion of solver collaboration was first elaborated in (Monfroy, 1996), page 4.

[14] In fact, on the finiteness of the approximate domain where an *approximate domain* A over a domain D is a subset of $\wp(D)$, with $D \in A$, closed under intersection.

in DecLic (Benhamou et al., 1997; Goualard et al., 1999), a CLP language that mixes Boolean, integer and real constraints in the framework of intervals. This system was shown to be fairly efficient on classical benchmarks but at the expense of decreasing the declarativity of the language as consequence of allowing the programmer to choose the best consistency to use for each constraint.

As well (Granvilliers, 2001) tackles the combination of interval techniques for solving non-linear systems. Granvilliers describes a cooperative strategy to combine the interval-based local consistencies methods (i.e., box and hull consistency) with the multi-dimensional interval Newton method and shows the efficiency of the main algorithm.

Another general scheme for solver cooperation is proposed in (Hofstedt, 2000). In this paper, domains are defined by using "Σ-Structures" in a sorted language and a constraint is a relation over an n-ary Cartesian product of the domains. As in our proposal, the combination of the solvers is achieved by means of the Cartesian product of the different domains. However, Hofstedt focuses on the interface between the solvers so that the complete system is a combination of this interface with the set of constraint systems each of which with its own associated solver for a specific domain. In contrast, in our proposal, it is the high level constraints that determine the possible cooperation that can occur between the domains and their solvers and these constraints may be defined by the user or system. The flexibility of these high level constraints implies that the solver interface defined by Hofstedt could be implemented in our system.

Another, but related form of cooperation is that described in the papers of Baader and Schulz. For instance (Baader and Schulz, 1995) provides an abstract framework to combine constraint languages and constraint solvers and focuses on ways in which different and independently defined solvers may be combined. In contrast to our proposal, they were not concerned with the constraint solving mechanism but with defining the properties that the structure supporting the framework has to satisfy to be suitable for the combination of constraint languages and solvers. More recently, Baader and Schulz present a general method for the combination of constraint systems (Baader and Schulz, 1998). This method is applicable to a kind of structures called *quasi-structures* that comprise very diverse structures such as algebras of (quotient) terms and rational trees, lists, sets, etc. This algorithm is a generalisation of another algorithm presented in (Baader and Schulz, 1996). The particularity of these algorithms is that they are extensions of previous algorithms developed with the aim of combining unification algorithms for equational theories where a unification algorithm can be seen as an instance of constraint solvers. Thus, as pointed out in (Kepser and Richts, 1999), these algorithms inherit the problems presented in the combination of equational theories: for instance a lack of practical use. In recent years, some proposals to overcome these problems have appeared.

Another interesting proposal has been recently reflected in the HAL system (de la Banda et al., 2001). This system supports the extension of existing solvers and the construction of hybrid ones. The proposal is noticeably distinct to ours. HAL provides semi-optional type, mode and determinism declarations for predicates and functions as well as a system of type classes over which constraint solvers'capabilities

are specified. A type class captures the notion of a type having an associated constraint solver. This concept of type class seems to be promising and its possible integration in our cooperative framework could be an issue of further work.

As it is shown in this section, cooperation may be understood from multiple perspectives. For more information the reader is referred to (Granvilliers et al., 2001) that provides a short introduction, with a number of important bibliography references, to present the basic foundations of solver cooperation.

5.7 Concluding Remarks

In this chapter we have extended our basic framework to enable a mechanism that ensures the flow of information between the computation domains. The extension is based on the definition of high level constraints as a relation over a domain constructed from a set of, possibly distinct, computation domains. In the resulting framework, solvers may be viewed as relations over one or more computation domains and thus solvers may be defined on very different domains and even one solver may be defined on several domains. As consequence, our mechanism for communication makes it possible for solvers to interact and, hence, cooperate.

We have also shown by a number of non-trivial and practical examples the flexibility and expressivity of the resulting framework.

Our cooperative framework also provides two very important characteristics that each cooperative system should hold:

1. *Type safety*: the erroneous cooperation of solvers is prevented in our framework. Observe that solvers can be freely connected by means of interval constraints using constraint operators defined on multiple domains. Therefore, only constraints that are valid in the framework with respect to the operator declarations are allowed in the formulation of problems.

2. *Transparency in the communication of solvers*: In our framework the cooperation is due to our defining the constraint operators on multiple domains and allowing high level constraints for specifying the intended propagation between the different domains. As already shown, both constraint operators and high level constraints are transparent to the user so that the cooperative schema is a glass box.

3. *Solver collaboration*, that is to say, solver combination plus solver cooperation.

5.8 Contributions

We present a novel mechanism for constraint cooperation over which the interaction of solvers is achieved by means of constraint operators defined on multiple domains and by the correct definition of the high level constraints. As consequence, constraint operators play the role of the interface between different solvers whereas high level

constraints play the role of pipes through which information is sent to and received from other solvers. To our knowledge, this cooperative approach is completely new.

From the user's point of view, the constraint operators and high level constraints are transparent, so that our cooperative approach is a glass box one. Moreover, to our knowledge, our framework (perhaps with the exception of CHR) is the only constraint setting that allows users to define their own interface between the solvers so that they can control the degree of cooperation (i.e., the direction in which information has to flow). Observe that this is not the case with other systems such as Prolog IV or CLP(BNR) in which the solvers are boxed in the systems and there is no possibility for integrating new ones at the user level.

Observe that, as consequence of integrating the whole cooperation process in one unique framework, the time spent in the coordination and synchronization of solvers (in a cooperative system constructed from independent solvers) is nill. This is another advantage of our framework.

Chapter 6

Interval Constraint Branching

Nothing is particularly hard if you divide it into small jobs.

Henry Ford 1863 - 1947

6.1 Introduction

To solve a CSP, we need to find an assignment of values to the variables such that all constraints are satisfied. A CSP can have many solutions; usually either any one or all of the solutions must be found. However, sometimes, because of the cost of finding all solutions, *partial* CSPs are used where the aim is just to find the best solution within fixed resource bounds. An example of a partial CSP is a *constraint optimisation problem* (COP) that assigns a value to each solution and tries to find an optimal solution (with respect to these values) within a given time frame.

In previous chapters we have described a generic interval constraint propagation schema to solve CSPs (i.e., a set of interval constraints defined on a set of lattice structure computation domains). Our schema removes inconsistent values from the initial domain of the variables that cannot be part of any solution. We have shown that the results are propagated through the whole constraint set and the process is repeated until a stable set is obtained. However, although our propagation schema will find a *most general solution* to the constraint store representing a CSP (see Theorem 4.56), it is not complete in the sense that it may not determine which values in these intervals are the correct answers to the problem.

For this reason, in this chapter, we propose a branching schema that is complementary to the constraint propagation schema already described. The combination of these two schemas forms an interval constraint solving framework that can be used for any set of domains which have the structure of a lattice, independently of their nature and, in particular, their cardinality. As consequence it can be used for most existing constraint domains (finite or continuous) and, as for the framework described in previous chapters, is also applicable to multiple domains and cooperative systems.

We also describe here some interesting properties that are satisfied by any instance of the branching schema and show that the operational procedures of many interval constraint systems (including cooperative systems) are instances of our branching schema.

6.1.1 Chapter Structure

This chapter is organised as follows. Section 6.2 defines some key concepts used in the chapter. Section 6.3 describes the main functions involved in interval constraint solving paying special attention to those required in the branching step. In Section 6.4, a generic schema for classical interval constraint solving on any set of lattices is developed and its main properties are stated. Section 6.5 extends this schema for partial constraint solving and then describes some more interesting properties. Section 6.6 provides an example illustrating the different ways in which a problem can be solved. As in the rest of the thesis, the chapter ends with a discussion about related work, some conclusions and a summary of the main contributions of the chapter.

6.2 Key Concepts

We continue to use L to denote any domain in $\mathcal{L}$, $X \in \wp_f(\mathcal{V}_\mathcal{L})$ the set of constrained variables, $\mathcal{C}_L^X$ the set of all interval constraints for L with constrained variables in X, $\mathcal{C}^X$ the interval constraint domain over X for L and $\mathcal{SS}^X$ the set of all simple stable constraint stores for X. Also $L_<$ denotes any totally ordered lattice in $\mathcal{L}$.

Notation. If $\{c_1, \ldots, c_n\} \in \mathcal{SS}^X$ and $i \in \{1 \ldots, n\}$, then

$$\{c_1, \ldots, c_n\}[c_i/c'] = \{c_1, \ldots, c_{i-1}, c', c_{i+1}, \ldots, c_n\}.$$

DEFINITION 6.1 *(Divisibility) Let $c = x \sqsubseteq \overline{s}, t$ be a consistent interval constraint in $\mathcal{C}_L^X$. Then, c is* divisible *if $s \neq_{L^s} t$ and* non-divisible *otherwise.*

Let $S \in \mathcal{SS}^X$ be a consistent constraint store. Then S is divisible *if there exists $c \in S$ such that c is divisible and* non-divisible *otherwise.*

Note that, by Definition 4.21, a non-divisible constraint has the form $x \sqsubseteq$ **[a,a]** that, as said in Definition 4.26, is a shorthand for $x =$ a where $x \in V_L$ and a $\in L$ for some $L \in \mathcal{L}$ (i.e., a non-divisible constraint may be viewed as an assignment of a constrained variable in a domain to a value belonging to that domain).

EXAMPLE 6.2 *Let $x, y \in V_{Integer}$, $r, w \in V_\Re$ and $S, S' \in \mathcal{SS}^{\{x,r\}}$. Then,*

$$x \sqsubseteq [\mathbf{1}, \mathbf{4}] \quad \textit{and} \quad r \sqsubseteq (\mathbf{1.0}, \mathbf{3.2}] \qquad \textit{are divisible;}$$
$$y \sqsubseteq [\mathbf{2}, \mathbf{2}] \quad \textit{and} \quad w \sqsubseteq [\mathbf{1.5}, \mathbf{1.5}] \qquad \textit{are non-divisible.}$$

Also

$$S = \{\ x \sqsubseteq [\mathbf{1}, \mathbf{1}],\ r \sqsubseteq [\mathbf{1.0}, \mathbf{1.0}]\ \} \quad \textit{is non-divisible;}$$
$$S' = \{\ x \sqsubseteq [\mathbf{1}, \mathbf{4}],\ r \sqsubseteq [\mathbf{1.0}, \mathbf{1.0}]\ \} \quad \textit{is divisible.}$$

More cases of non-divisibility can be detected by the equivalence of ranges on discrete domains shown in Section 4.4.4 on page 102.

EXAMPLE 6.3 *Consider $L = Integer$ and the consistent constraint $c = x \sqsubseteq (\mathbf{1}, \mathbf{2}]$. Then c is non-divisible since the range $(\mathbf{1}, \mathbf{2}]$ is equivalent to the range $[\mathbf{2}, \mathbf{2}]$ in R^s_L and the constraint $c = x \sqsubseteq [\mathbf{2}, \mathbf{2}]$ is non-divisible.*

PROPOSITION 6.4 *Let $X \in \wp_f(V_\mathcal{L})$.*

(1) Let also $c, c' \in \mathcal{C}^X_L$ such that $c \prec_{\mathcal{C}^X_L} c'$. Then, if c is consistent, c' is divisible.

(2) Let also $S, S' \in \mathcal{SS}^X$ such that $S \prec_s S'$. Then, if S is consistent, S' is divisible.

PROOF 6.5 *We prove the cases separately.*

Case (1). *Suppose that $c = x \sqsubseteq r$ and $c' = x \sqsubseteq r'$, where $r = \overline{s}, t$ and $r' = \overline{s'}, t'$. By hypothesis c is consistent and thus, by Definition 4.26 on page 96, r is consistent and also $r \prec_{R^s_L} r'$. By Proposition 4.23, r' is consistent and thus, again by Definition 4.26, c' is consistent. Therefore, by Definition 4.21*

$$s \preceq_{L^s} t \wedge s' \preceq_{L^s} t'.$$

Moreover, as $r \prec_{R^s_L} r'$, by Definition 4.19,

$$\overline{s} \prec_{\overline{L^s}} \overline{s'} \wedge t \preceq_{L^s} t' \vee$$
$$\overline{s} \preceq_{\overline{L^s}} \overline{s'} \wedge t \prec_{L^s} t'.$$

It follows by the duality principle for lattices (see Page 83) that

$$s' \prec_{L^s} t'.$$

Therefore, by Definition 6.1, c' is divisible.

Case (2). *By hypothesis S is consistent and thus by Proposition 4.37 on page 100, S' is consistent. By Definition 4.30, for all $c \in S$, c is consistent and also, by Definition 4.33, there exists $c' \in S'$ and $c \in S$ such that $c \prec_{\mathcal{C}^X} c'$. By Proposition 6.4(1), c' is divisible. Therefore, by Definition 6.1, S' is divisible.*

□

In Section 4.4.5 on page 103 we defined the concept of *solution* for a constraint store. A solution was defined to be a consistent stable store that produces no more constraint narrowing by constraint propagation. In this section we redefine this solution concept to capture the usual meaning of a solution as an assignment of values to variables that satisfies all the constraints. So as to distinguish the solution defined in Section 4.4.5 from the concept defined in this chapter, we use the term *solution* to refer the concept already defined and the term *authentic solution* to refer the new concept defined in this chapter.

DEFINITION 6.6 *(Authentic solution) Let $C \in \wp_f(\mathcal{C}^X)$ be a constraint store for X and $R \in \mathcal{SS}^X$. Then, R is an* authentic solution for C *if R is both non-divisible and a solution for C.*

$R' \in \mathcal{SS}^X$ is a partial solution for C *if there exists an authentic solution R'' for C such that $R'' \prec_s R'$. In this case we say that R'* covers *R''.*

EXAMPLE 6.7 *Consider the operators $+$ and $-$ for $L = Integer$ as defined in Example 4.11 on page 88, $x, y \in V_{Integer}$, $X = \{x, y\}$, $C \in \wp_f(\mathcal{C}^X)$ where*

$$\begin{aligned} C \;=\; \{\; & x \sqsubseteq [0, max(y) - [1, \\ & y \sqsubseteq [1 \overline{+} min(x), 100] \;\} \end{aligned}$$

and $S, S' \in \mathcal{SS}^X$ where

$$\begin{aligned} S \;&=\; \{\; x \sqsubseteq [\mathbf{1}, \mathbf{4}],\; y \sqsubseteq [\mathbf{2}, \mathbf{5}] \;\}, \\ S' \;&=\; \{\; x \sqsubseteq [\mathbf{1}, \mathbf{1}],\; y \sqsubseteq [\mathbf{3}, \mathbf{3}] \;\}. \end{aligned}$$

Then, S is a solution (and also a partial solution) for C whereas S' is an authentic solution for C.

The set of all authentic solutions for C is denoted as $Sol_a(C)$.

DEFINITION 6.8 *(Constraint store stack) Let $P = (S_1, \ldots, S_\ell)$ be any (possibly empty) sequence where $S_i \in \mathcal{SS}^X$ for $1 \leq i \leq \ell$ and $\ell \geq 0$. Then P is a* constraint store stack for X *if the operation $push/2$ over P is defined for any $S \in \mathcal{SS}^X$ as follows*

$$\begin{aligned} &\mathbf{P}recondition : \{\; P = (S_1, \ldots, S_\ell) \;\} \\ &\quad push(P, S) \\ &\mathbf{P}ostcondition : \{\; P = (S_1, \ldots, S_\ell, S_{\ell+1}),\; S_{\ell+1} = S \text{ and } P \in Stack(X) \;\}. \end{aligned}$$

where $Stack(X)$ is the set of all constraint store stacks for X, and the operation $top/1$ over P is defined as:

$$\begin{aligned} &\mathbf{P}recondition : \{P = (S_1, \ldots, S_\ell) \text{ and } \ell > 0\} \\ &\quad top(P) = S \\ &\mathbf{P}ostcondition : \{S = S_\ell\}. \end{aligned}$$

Let $P' = (S'_1, \ldots, S'_{\ell'})$ be another constraint store stack for X. Then $P \preceq_p P'$ if and only if for all $S_i \in P$ $(1 \leq i \leq \ell)$, there exists $S'_j \in P'$ $(1 \leq j \leq \ell')$ such that $S_i \preceq_s S'_j$. In this case we say that P' covers *P.*

6.3 The Branching Process

In Section 2.2.3 it was shown that branching often involves two steps of choice usually called *variable ordering* and *value ordering*. The first step selects a constrained variable and the second one splits the domain associated to the selected variable in order to introduce a choice point. In this section we explain these choice steps by describing the main functions that define them.

The *selecting function* provides a schematic heuristic for variable ordering.

DEFINITION 6.9 *(Selecting function) Let* $S = \{c_1, \ldots, c_n\} \in \mathcal{SS}^X$. *Then*

$$choose :: \{S \in \mathcal{SS}^X \mid S \text{ is divisible}\} \to \mathcal{C}^X$$

is called a selecting function for X *if* $choose(S) = c_j$ *where* $1 \leq j \leq n$ *and* c_j *is divisible.*

EXAMPLE 6.10 *Suppose that* $X = \{x_1, \ldots, x_n\}$ *is a set of variables constrained respectively in* $L_1, \ldots, L_n \in \mathcal{L}$ *and that* $S = \{c_1, \ldots, c_n\} \in \mathcal{SS}^X$ *is any divisible constraint store for* X *where for all* $i \in \{1, \ldots, n\}$, c_i *is the simple interval constraint in* S *with constrained variable* x_i. *Here is a naive strategy that selects the "left-most" divisible interval constraint in* S.

$$\begin{aligned}
&\mathbf{P}recondition : \{S = \{c_1, \ldots, c_n\} \in \mathcal{SS}^X \text{ is divisible}\} \\
&\quad choose_{naive}(S) = c_j \\
&\mathbf{P}ostcondition : \{j \in \{1, \ldots, n\},\ c_j \text{ is divisible and} \\
&\qquad\qquad \forall i \in \{1, \ldots, j-1\} : c_i \text{ is non-divisible}\}.
\end{aligned}$$

When branching, some interval constraints need to be partitioned, into two or more parts, so as to introduce a choice point. We define a *splitting function* which provides a heuristic for value ordering.

DEFINITION 6.11 *(Splitting function) Let* $L \in \mathcal{L}$ *and* $k > 1$. *Then*

$$split_L :: \mathcal{C}_L^X \to \underbrace{\mathcal{C}_L^X \times \ldots \times \mathcal{C}_L^X}_{k \text{ times}}$$

is called a k-ary splitting function for L *if, for all* $c \in \mathcal{C}_L^X$, *with* c *divisible, this function is defined* $split_L(c) = (c_1, \ldots, c_k)$ *such that the following properties hold:*

$$\begin{aligned}
&Completeness : \ \forall c' \prec_{\mathcal{C}_L^X} c \text{ with } c' \text{ non-divisible}, \exists i \in \{1, \ldots, k\} \,.\, c' \preceq_{\mathcal{C}_L^X} c_i. \\
&Contractance : \ c_i \prec_{\mathcal{C}_L^X} c,\ \forall i \in \{1, \ldots, k\}.
\end{aligned}$$

EXAMPLE 6.12 *Let $X = \{i, b, r, s\}$ be a set of variables where $i \in V_{Integer}$, $b \in V_{Bool}$, $r \in V_{\Re}$ and $s \in V_{Set\ Integer}$ and let*[1] *$i \sqsubseteq [\mathbf{a}, \mathbf{a}']$, $b \sqsubseteq [false, true]$, $r \sqsubseteq \{\mathbf{c}, \mathbf{d}\}$ and $s \sqsubseteq \{\mathbf{e}, \mathbf{f}\}$ be divisible interval constraints in $\mathcal{C}^X$ where $\mathbf{a}, \mathbf{a}' \in Integer$, $\mathbf{c}, \mathbf{d} \in \Re$ and $\mathbf{e}, \mathbf{f} \in Set\ Integer$. Then, the following functions are binary splitting functions respectively for the domains Integer, Bool, $\Re$ and Set Integer*

$$split_{Integer}(\ x \sqsubseteq [\mathbf{a}, \mathbf{a}']) = (\ x \sqsubseteq [\mathbf{a}, \mathbf{a}],\ x \sqsubseteq [\mathbf{a} + \mathbf{1}, \mathbf{a}']\ ),$$
$$split_{Bool}(\ b \sqsubseteq [false, true]\ ) = (\ b \sqsubseteq [false, false], b \sqsubseteq [true, true]\ ),$$
$$split_{\Re}(r \sqsubseteq \{\mathbf{c}, \mathbf{d}\}) \ = \ (\ r \sqsubseteq \{\mathbf{c}, \mathbf{c}'),\ r \sqsubseteq [\mathbf{c}', \mathbf{d}\}\ ),$$
$$split_{Set\ \ Integer}(s \sqsubseteq \{\mathbf{e}, \mathbf{f}\}) = (\ s \sqsubseteq \{\mathbf{e}, \mathbf{f} \backslash \mathbf{g}],\ s \sqsubseteq [\mathbf{e} \cup \mathbf{g}, \mathbf{f}\}\ ).$$

Here, $split_{Integer}$ is a naive enumeration strategy in which values are chosen from left to right; $split_{Bool}$ divides the only divisible Boolean interval constraint into the two non-divisible Boolean interval constraints; $split_{\Re}$ computes the mid point $\mathbf{c}' = \frac{\mathbf{c}+\mathbf{d}}{\mathbf{2.0}}$ of the interval $[\mathbf{c}, \mathbf{d}]$; and $split_{Set\ \ Integer}$ is a valid splitting function for the domain of sets of integers if we define $\mathbf{g} = \{\mathbf{l}\}$ and $\mathbf{l} \in \mathbf{f} \backslash \mathbf{e}$.

LEMMA 6.13 *Let choose/1 be a selecting function for X, $C \in \wp_f(\mathcal{C}^X)$, $S = (c_1, \ldots, c_n) \in \mathcal{SS}^X$ a divisible constraint store, $c_j = choose(S)$, $c_j \in \mathcal{C}_L^X$ for some $L \in \mathcal{L}$, $split_L/1$ a k-ary splitting function for L and $(c_{j1}, \ldots, c_{jk}) = split_L(c_j)$. Then*

(a) $\forall i \in \{1, \ldots, k\} : S[c_j/c_{ji}] \prec_s S$;

(b) if $S' \in Sol_a(C)$ and $S' \prec_s S$, then

$$\exists i \in \{1, \ldots, k\} : S' \preceq_s S[c_j/c_{ji}].$$

PROOF 6.14 *We prove the cases separately.*

Case (a). *By Definition 6.9, c_j is divisible and, by the contractance property shown in Definition 6.11, for all $i \in \{1, \ldots, k\}$ $c_{ji} \prec_{\mathcal{C}_L^X} c_j$. Therefore, by Definition 4.33, for all $i \in \{1, \ldots, k\}$ $S[c_j/c_{ji}] \prec_s S$.*

Case (b). *By Definition 6.6, $S' \in \mathcal{SS}^X$. Suppose that c_j is constrained on some variable $x \in V_L$ ($x \in X$) and let c'_j be the simple interval constraint for x in S'. Thus, by Definition 4.33 on page 100 $c'_j \preceq_{\mathcal{C}_L^X} c_j$. Moreover, by Definition 6.6 S' is non-divisible and thus by Definition 6.1, c'_j is non-divisible. Also by Definition 6.9 c_j is divisible so that $c'_j \neq c_j$ and thus $c'_j \prec_{\mathcal{C}_L^X} c_j$. As consequence, by the completeness property of the splitting functions shown in Definition 6.11,*

$$\exists i \in \{1, \ldots, k\}\ .\ c'_j \preceq_{\mathcal{C}_L^X} c_{ji} \tag{6.1}$$

Therefore, again by Definition 4.33, $\exists i \in \{1, \ldots, k\}$ such that $S' \preceq_s S[c_j/c_{ji}]$.

□

[1] Observe that in the integer and Boolean domains only intervals with close brackets are considered since by the equivalence of ranges on discrete domains shown in Section 4.4.4 on page 102 open brackets can always be transformed in close brackets e.g., $x \sqsubseteq (\mathbf{1}, \mathbf{8})$ is equivalent to $x \sqsubseteq [\mathbf{2}, \mathbf{7}]$. Note also that in the Boolean domain there is just one unique case of divisible interval constraint (i.e., $b \sqsubseteq [false, true]$) and thus only this case is considered in the definition of $split_{Bool}$.

6.3.1 The Precision Map as a Normalisation Rule

The precision map already described in Section 4.5.2 also provides a way to normalise the selecting functions (i.e., the variable ordering) when the constraint system supports multiple domains.

EXAMPLE 6.15 *The well known* first fail principle *(see Example 2.3) chooses the variable constrained with the smallest domain. However, in systems supporting multiple domains it is not always clear which is the smallest domain (particularly if there are several infinite domains). In our framework, one way to "measure" the size of the domains is to use the precision map defined on each computation domain.*

For instance, suppose that $X = \{x_1, \ldots, x_n\}$ is a set of variables constrained, respectively, in $L_1, \ldots, L_n \in \mathcal{L}$ and that $S = \{c_1, \ldots, c_n\} \in \mathcal{SS}^X$ is any divisible constraint store for X where for each $i \in \{1, \ldots, n\}$, c_i is the simple interval constraint in S with constrained variable x_i. Here the first fail principle can be emulated by defining choose/1 to select the interval constraint with the smallest precision[2]*. We denote this procedure by $choose_{ff}$.*

$$\begin{aligned}
&\mathbf{P}recondition : \{S = \{c_1, \ldots, c_n\} \in \mathcal{SS}^X \text{ is divisible}\}\\
&\quad choose_{ff}(S) = c_j\\
&\mathbf{P}ostcondition : \{j \in \{1, \ldots, n\}, c_j \text{ is divisible and}\\
&\quad \forall i \in \{1, \ldots, n\} \backslash \{j\} : c_i \text{ divisible} \implies precision_{L_j}(c_j) \leq_{\Re\mathcal{I}} precision_{L_i}(c_i)\}.
\end{aligned}$$

6.4 Branching in Interval Constraint Solving

Figure 6.1 shows a generic schema for solving completely the interval constraints. This schema is complementary to that shown in Section 4.5.

We continue to use L to denote any domain in $\mathcal{L}$, $X \in \wp_f(\mathcal{V}_\mathcal{L})$ the set of constrained variables, $\mathcal{C}^X$ the set of all interval constraints domain for X and $\mathcal{SS}^X$ the set of all simple stable constraint stores for X.

The schema requires the following parameters: a finite set $C \in \wp_f(\mathcal{C}^X)$ of interval constraints to be solved, a constraint store $S \in \mathcal{SS}^X$ and a bound $p \in \Re\mathcal{I}$. The schema requires a non-negative real bound α and also extends the operational schema $solve_\varepsilon/2$ described in Section 4.5.1. In the following we state some more properties on the propagation schema $solve_\varepsilon/2$.

LEMMA 6.16 *(More properties of $solve_\varepsilon/2$) Let $C \in \wp_f(\mathcal{C}^X)$, $S, S^f \in \mathcal{SS}^X$ and $\varepsilon \in \Re^+ \cup \{0.0\}$. Suppose that S^f is the value of the constraint store S after a terminating execution of $solve_\varepsilon(C, S)$. Then,*

(a) $S^f \preceq_s S$;

[2] It is straightforward to include more conditions e.g., if c_i, c_k, c_j have the same (minimum) precision, the "left-most" domain can be chosen i.e., $c_{minimum(i,k,j)}$.

(b) $\forall R \in Sol_a(C \cup S) : R \preceq_s S^f$;

(c) If $Sol_a(C \cup S)$ *is not empty and* S^f *is non-divisible then* $S^f \in Sol_a(C \cup S)$;

(d) If $\varepsilon = 0.0$ *and* S^f *is non-divisible then* $S^f \in Sol_a(C \cup S)$.

PROOF 6.17 *In the following, let* S_0 *be the initial value of* S *and* $C = C \cup S_0$. *Suppose that the procedure terminates after* k *iterations of the repeat loop (i.e.,* $S_k = S^f$*) and that, for each* i *where* $1 \leq i \leq k$, S_i *is the value of the constraint store* S *at step (5) of the schema shown in Section 4.5.1 on page 108, after completing* i *iterations of the repeat loop.*

Now we prove the cases separately.

Case (a).

We show by induction on i, *that after* $i \geq 0$ *iterations of the* repeat *loop*

$$S_i \preceq_s S_0. \tag{6.2}$$

It follows that after k *iterations* $S_k \preceq_s S_0$ *and thus* $S^f \preceq_s S_0$.

The base case when $i = 0$ *is obvious. For the inductive step, suppose that there are at least* $i > 0$ *iterations of the repeat loop and that, after* $i-1$ *steps, we have* $S_{i-1} \preceq_s S_0$. *Then, by Line 4,*

$$S_{i-1} \cup C' \mapsto S_i,$$

It follows from Proposition 4.35 on page 100 that $S_i \preceq_s S_{i-1}$. *Therefore by the inductive hypothesis* $S_i \preceq_s S_0$.

Case (b).

Let $R \in Sol_a(C \cup S)$. *By Definition 6.6,* R *is a solution for* $C \cup S$. *From here, following the same reasoning as in the proof of Theorem 4.56 on page 109 we obtain that*[3] $R \preceq_s S_k$. *As consequence* $R \preceq_s S^f$.

Case (c).

Let $R \in Sol_a(C \cup S)$. *By Case 6.16(b),* $R \preceq_s S^f$. *Suppose that* $R \prec_s S^f$. *By Definition 6.6,* R *is a solution for* $C \cup S$ *and, by Definition 4.46,* R *is consistent. Thus, by Proposition 6.4(2),* S^f *is divisible which contradicts the hypothesis. As consequence,* $R = S^f$ *and thus* $S^f \in Sol_a(C \cup S)$.

Case (d).

By Definition 6.6, S^k *(i.e.,* S^f*) is consistent and thus the procedure* $solve_\varepsilon(C, S)$ *terminates because*

$$precision(S_{k-1}) - precision(S_k) \leq (0.0, 0) \tag{6.3}$$

By Line 4, in the k*-th iteration,*

$$C \rightsquigarrow^{S_{k-1}} C'; \tag{6.4}$$

$$S_{k-1} \cup C' \mapsto S_k. \tag{6.5}$$

[3] Observe that the outline to prove (4.16) is exactly the same for the procedure *solve*/2 and the procedure $solve_\varepsilon/2$.

Then, by (6.5) *and Proposition 4.35 on page 100* $S_k \preceq_s S_{k-1}$. *As consequence, from* (6.3) *and Proposition 4.65,* $S_k = S_{k-1}$. *By* (6.4), (6.5) *and Definition 4.46,* S_k *is a solution for* C *(i.e.,* $C \cup S_0$*). Therefore, as* S_k *is non-divisible, by Definition 6.6,* $S_k \in Sol_a(C \cup S)$.

□

Property (a) ensures that the propagation procedure never gains values, property (b) guarantees that no solution covered by a constraint store is lost in the propagation process and properties (c) and (d) guarantee the computed answers are correct[4].

There are a number of values and subsidiary procedures that are assumed to be defined externally to the main branch procedure shown in Figure 6.1:

- a selecting function *choose*/1 for X;
- a k-ary splitting function $split_L$ for each domain $L \in \mathcal{L}$ (for some integer $k > 1$);
- a precision map for each $L \in \mathcal{L}$;
- a constraint store stack P for X.

It is assumed that the external procedures have an implementation that terminates for all possible values.

Before stating the main properties of the schema shown in Figure 6.1, we define some concepts that will be useful to prove them.

A path $q \in (Natural \backslash \{0\})^*$ is any finite sequence of (non-zero) natural numbers. The empty path is denoted by ϵ, whereas $q.i$ denotes the path obtained by concatenating the sequence formed by the natural number $i \neq 0$ with the sequence of the path q. The length of the sequence q is called the *length* of the path q.

Given a tree, we label the nodes by the paths to the nodes. The root node is labelled ϵ. If a node with label q has k children, then they are labelled, from left to right, $q \,.\, 1, \ldots, q \,.\, k$.

DEFINITION 6.18 *(Search tree for* $branch_\alpha(C, S, p)$*). Let* $X \in \wp_f(\mathcal{V}_\mathcal{L})$, $S \in \mathcal{SS}^X$, $C \in \wp_f(\mathcal{C}^X)$, $\alpha \in \Re^+ \cup \{0.0\}$ *and* $p \in \Re\mathcal{I}$. *The search tree for* $branch_\alpha(C, S, p)$ *is a tree that has* S *at the root node and, as children, has the search trees for the recursive executions of* $branch_\alpha/3$ *as consequence of reaching Line 8 of Figure 6.1.*

Given a search tree for $branch_\alpha(C, S, p)$*, we say that* $S_\epsilon = S$ *is the constraint store and* $p_\epsilon = p$ *the precision at the root node* ϵ. *Let* S_q *be the constraint store and* p_q *the precision at a node* q. *If* q *has* $k > 0$ *children* $q \,.\, 1, \ldots, q \,.\, k$, *then* S_q *is consistent and, if* S_q^f *is the constraint store* S_q *after a terminating execution of* $solve_\varepsilon(C, S_q)$, *then* S_q^f *is divisible so that* $choose(S_q^f) = c_j$ *(for some* $c_j \in \mathcal{C}_{L_j}^X$ *and* $L_j \in \mathcal{L}$*) and, for some* $k > 0$, $split_{L_j}(c_j) = (c_{j1}, \ldots, c_{jk})$. *Then we say that* $S_{q.i} = S_q^f[c_j/c_{ji}]$ *is the constraint store and* $p_{q.i} = precision(S_q^f)$ *the precision at node* $q \,.\, i$, *for* $i \in \{1, \ldots, k\}$.

[4]Theorem 4.56 assures that, if a solution exist, the final state of constraint store S contains the most general solution but not an authentic solution.

procedure $branch_\alpha(C, S, p)$
begin
  $solve_\varepsilon(C, S)$; (1)
  if S is consistent then (2)
    if (S is non-divisible or $p < \top_{\Re\mathcal{I}}$ and $p - precision(S) \leq (\alpha, 0)$) then (3)
      $push(P, S)$; (4)
    else (5)
      $c_j \leftarrow choose(S)$; (6)
      $(c_{j1}, \ldots, c_{jk}) \leftarrow split_{L_j}(c_j)$, where $c_j \in \mathcal{C}^X_{L_j}$ and $L_j \in \mathcal{L}$; (7)
      $branch_\alpha(C, S[c_j/c_{j1}], precision(S)) \;\vee$
      $\ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \;\vee$
      $branch_\alpha(C, S[c_j/c_{jk}], precision(S));$ } %% Choice Points (8)
    endif;
  endif;
end.

Figure 6.1: $branch_\alpha/3$: a generic schema for interval constraint solving

THEOREM 6.19 *(Properties of the $branch_\alpha/3$ schema) Let $C \in \wp_f(\mathcal{C}^X)$, $S \in \mathcal{SS}^X$, $\varepsilon, \alpha \in \Re^+ \cup \{0.0\}$ and $p = \top_{\Re\mathcal{I}}$. Then, the following properties are guaranteed:*

1. Termination*: if $\alpha > 0.0$ and the procedure $solve_\varepsilon/2$ terminates for all values*[5] *then $branch_\alpha(C, S, p)$ terminates;*

2. Completeness: *if $\alpha = 0.0$ and the execution of $branch_\alpha(C, S, p)$ terminates, then the final state for the stack P contains all the authentic solutions for $C \cup S$;*

3. Approximate completeness: *if the execution of $branch_\alpha(C, S, p)$ terminates and $R \in Sol_a(C \cup S)$, then the final state for the stack P contains either R or a partial solution R' that covers R.*

4. Correctness: *if $\alpha = 0.0$ and $\varepsilon = 0.0$, the stack P is initially empty and the execution of $branch_\alpha(C, S, p)$ terminates with R in the final state of P, then $R \in Sol_a(C \cup S)$.*

5. Approximate correctness or control on the result precision: *If P_{α_1} and P_{α_2} are non-empty constraint store stacks for X resulting from any terminating execution of $branch_\alpha(C, S, p)$ (where initially P is empty) when α has the values α_1 and*

[5] Observe that termination of this procedure is always guaranteed if $\varepsilon > 0.0$ -See Theorem 4.67.

α_2, respectively, and $\alpha_1 < \alpha_2$ then

$$P_{\alpha_1} \preceq_p P_{\alpha_2}.$$

(In other words, the set of (possibly partial) solutions in the final state of the stack is dependent on the value of α in the sense that the lower α, the better the set of solutions.)

Observe that, just as for the bound ε in the $solve_\varepsilon/2$ procedure, the bound α also guarantees termination and allows the precision of the results to be controlled.

In the following, each property stated in Theorem 6.19 is proved independently.

PROOF 6.20 *(Property (1). Termination) In the following, we show that the search tree for $branch_\alpha(C, S, p)$ is finite so that the procedure effectively terminates.*

Let $S_\epsilon = S$ and $p_\epsilon = p$. If the search tree for $branch_\alpha(C, S_\epsilon, p_\epsilon)$ has only one node then the procedure terminates. Otherwise, the root node ϵ has k children with constraint stores S_i where $i \in \{1, \ldots, k\}$ and $S_i = S_\epsilon^f[c_j/c_{ji}]$. By Lemma 6.13(a) and Lemma 6.16(a), for all $i \in \{1, \ldots, k\}$, $S_i \prec_s S_\epsilon$ and, by Proposition 4.65, $precision(S_i) <_{\Re\mathcal{I}} precision(S_\epsilon)$. Then, $precision(S_i) <_{\Re\mathcal{I}} \top_{\Re\mathcal{I}}$. Suppose now that $precision(S_i) = (\top_\Re, n)$ for some $n \in Integer$. Then the test in Line 2 $p_i - precision(S_i) \leq_{\Re\mathcal{I}} (\alpha, 0)$ holds and the node containing S_i has no children. Otherwise,

$$p_i - precision(S_i) >_{\Re\mathcal{I}} (\alpha, 0) \tag{6.6}$$

and there exists some constant $\ell \in \Re$ such that

$$precision(S_i) <_{\Re\mathcal{I}} (\ell \times \alpha, 0).$$

We show by induction on the length $j \geq 1$ of a path q in the search tree that

$$precision(S_i) - precision(S_q^f) \geq_{\Re\mathcal{I}} \big((j-1) \times \alpha, 0\big).$$

It follows that $j \leq \ell$ and that, all paths have length $\leq \ell + 1$ (since the second condition in Line 3 of Figure 6.1 holds) and thus there are no infinite branches.

The base case when $j = 1$ follows from (6.6). Suppose next that $j > 1$ and that the hypothesis holds for a path q of length $j-1$. Let $q\,.\,i_q$ be a child of q of length j. Then, by the condition in Line 3 of the if sentence,

$$p_{q.i_q} - precision(S_{q.i_q}^f) >_{\Re\mathcal{I}} (\alpha, 0),$$

However, by the inductive hypothesis,

$$precision(S_i) - precision(S_q^f) \geq_{\Re\mathcal{I}} \big((j-2) \times \alpha, 0\big)$$

so that, as $precision(S_q^f)$ *is* $p_{q.i_q}$*,*

$$precision(S_i) - precision(S_{q.i_q}) \geq_{\Re\mathcal{I}} (\alpha, 0) + \big((j-2) \times \alpha, 0\big) = \big((j-1) \times \alpha, 0\big).$$

□

PROOF 6.21 *(Property (2). Completeness) Let* $R \in Sol_a(C \cup S)$*. Then,* R *is non-divisible and consistent by Definitions 6.6 and 6.1. By Lemma 4.47 on page 103,* $R \preceq_s S_\epsilon$ *and, by Lemma 6.16(b),* $R \preceq_s S_\epsilon^f$*. If* $R = S_\epsilon^f$ *then tests in Lines 2-3 hold and* R *is pushed on the stack* P*. Otherwise,* $R \prec_s S_\epsilon^f$*. By Proposition 6.4(2),* S_ϵ^f *is divisible, (and thus by Definition 6.1 consistent). As* $p_\epsilon = \top_{\Re\mathcal{I}}$*, the condition in Line 3 does not hold and node* ϵ *has* k *children. By Lemma 6.13(a) and Lemma 6.16(a), for any* q *of length* $m \geq 1$ *and* $i_q \in \{1, \ldots, k\}$*,* $S_{q.i_q}^f \prec_s S_q^f$*. By Proposition 4.65,* $precision(S_q^f) - precision(S_{q.i_q}^f) > (0.0, 0)$*. Thus the condition* $p_{q.i_q} - precision(S_{q.i_q}^f) \leq (\alpha, 0)$ *in Line 3 never holds. It follows that all the branches in the tree terminate either with an inconsistent store (because test in Line 2 does not hold) or with a non-divisible store (that is also consistent by Definition 6.1) as result of holding tests in Lines 2 and 3. Now, we show by induction on the length* $j \geq 1$ *of a path* q *in the search tree that*

$$R \prec_s S_q^f \implies \exists i_q \in \{1, \ldots, k\} : R \preceq_s S_{q.i_q}^f. \tag{6.7}$$

By hypothesis, the procedure terminates so that the search tree is finite. It follows that there exists some path $p = q \,.\, q'$ *with a finite length* $l \geq j$ *such that* $R = S_p^f$*. Thus,* S_p^f *is non-divisible (and, by Definition 6.1 is consistent) and hence tests in Lines 2 and 3 hold so that* R *is pushed on the stack* P*.*

In the base case, when $j = 1$*,* $S_i = S_\epsilon^f[c_j/c_{ji}]$ *(*$i \in \{1, \ldots, k\}$*). By Lemma 6.13(b) and Lemma 6.16(b),* $\exists i \in \{1, \ldots, k\} : R \preceq_s S_i^f$*. Suppose next that* $j > 1$ *and that the hypothesis holds for a path* q *of length* $j - 1$ *so that* $R \preceq_s S_q^f$*. If* $R \prec_s S_q^f$ *then, by Proposition 6.4(2),* S_q^f *is divisible (and thus consistent by Definition 6.1) so that the node* S_q^f *has* k *children. Therefore, by Lemma 6.13(b) and Lemma 6.16(b),* $\exists i_q \in \{1, \ldots, k\} : R \preceq_s S_{q.i_q}^f$*.*

□

PROOF 6.22 *(Property (3). Approximate completeness) Let* $R \in Sol_a(C \cup S)$*. By Lemma 4.47 on page 103,* $R \preceq_s S_\epsilon$*. Since the procedure terminates, as shown in proof of Theorem 6.19(1), all paths in the search tree have length* $\leq \ell + 1$*. Therefore, as shown in proof of Theorem 6.19(2) (completeness proof), by following* (6.7)*, there must exists some path* q *with no children and length* $j \geq 1$ *such that* $R \preceq_s S_q^f$*. If* $R = S_q^f$ *then* R *is put on the stack since, by Definition 6.1,* R *is consistent so that tests in Lines 2 and 3 hold. Otherwise, as shown in termination proof, the node* S_q^f *has no more children since the test* $p_q - precision(S_q^f) \leq_{\Re\mathcal{I}} (\alpha, 0)$ *holds and* S_q^f *is put on the stack. As* $R \preceq_s S_q^f$*, by Definition 6.6, either* $S_q^f \in Sol_a(C \cup S)$ *or is a partial solution for* $C \cup S$ *that covers* R*.*

□

PROOF 6.23 *(Property (4). Correctness) Let $R \in P$ after executing $branch_\alpha(C, S, p)$. As shown in completeness proof, if $\alpha = 0.0$ the test $p_q - precision(S_q^f) \leq (\alpha, 0)$ never holds, for all path q (in the search tree) of length $m \geq 1$ (observe also that Line 3 is never satisfied when $q = \epsilon$ since $p_\epsilon \not< \top_{\Re\mathcal{I}}$). Therefore, R is in P because there exists a path q where $S_q^f = R$ and the tests in Lines 2 and 3 hold. Thus, R is consistent and non-divisible and by Lemma 6.16(4), $R \in Sol_a(C \cup S_q)$.*

By induction on the length of the path q it is straightforward to prove that $S_q \preceq_s S_\epsilon$. Now we prove that if $R \in Sol_a(C \cup S_q)$ then $R \in Sol_a(C \cup S_\epsilon)$. By Definition 6.6 R is a solution for $C \cup S_q$ and thus by Definition 4.46 on page 103,

$$C \cup S_q \rightsquigarrow^R C'$$
$$R \cup C' \mapsto R.$$

As shown in proof of Lemma 4.47 on page 103, $C' = C_1 \cup S_q$ where $C \rightsquigarrow^R C_1$. Moreover, $C \cup S_\epsilon \rightsquigarrow^R C''$ where $C'' = C_1 \cup S_\epsilon$. Since $S_q \preceq_s S_\epsilon$, by Definition 4.31 on page 99,

$$R \cup C' \mapsto R \implies R \cup C'' \mapsto R.$$

Therefore

$$C \cup S_\epsilon \rightsquigarrow^R C''$$
$$R \cup C'' \mapsto R.$$

Thus, by Definition 4.46, R is as solution for $C \cup S$ and, by Definition 6.6, $R \in Sol_a(C \cup S_\epsilon)$.

□

PROOF 6.24 *(Property (5). Approximate correctness or control on the precision result) Suppose that $R \in P_{\alpha_1}$. Then R is consistent and there exists a path q of length $m \geq 0$ such that $S_q^f = R$ and S_q^f was pushed on the stack because the test in Line 3 holds but, for all proper prefixes of q, it does not hold. Thus either S_q^f is non-divisible or*

$$p_q < \top_{\Re\mathcal{I}} \text{ and } p_q - precision(S_q^f) \leq (\alpha_1, 0).$$

In addition, for all proper prefixes q_1 of q, $S_{q_1}^f$ is divisible and, either

$$p_{q_1} = \top_{\Re\mathcal{I}} \text{ or } p_{q_1} - precision(S_{q_1}^f) > (\alpha_1, 0).$$

Since $\alpha_1 \leq \alpha_2$, we also have

$$p_q < \top_{\Re\mathcal{I}} \text{ and } p_q - precision(S_q^f) \leq (\alpha_2, 0).$$

Let q' be the smallest prefix of q such that

$$p_{q'} < \top_{\Re\mathcal{I}} \text{ and } p_{q'} - precision(S^f_{q'}) \leq (\alpha_2, 0).$$

Then $S^f_{q'}$ is in P_{α_2}. By repetitive application of Lemmas 6.13(a) and 6.16(a) it is straightforward to prove that $S^f_q \preceq_s S^f_{q_1}$ for all proper prefix q_1 of q. Thus, $S^f_q \preceq_s S^f_{q'}$. Therefore, as the choice of $R \in P_{\alpha_1}$ was arbitrary, by Definition 6.8, $P_{\alpha_1} \preceq_p P_{\alpha_2}$.

□

Instances. Throughout this document we have already provided a number of instances of the branch schema including the integers, reals and sets. These illustrate that this schema can be used for any set of computation domains for which a splitting function and precision map are defined.

6.5 Solving Optimisation Problems

The schema in Figure 6.1 can be adapted to solve COPs by means of three new subsidiary functions.

DEFINITION 6.25 *(Subsidiary functions and values) Let $L_< \in \mathcal{L}$ be a totally ordered domain*[6]*. Then we define*

- *a* cost function*, $fcost :: \mathcal{SS}^X \to L_<$;*
- *an* ordering relation*, $\diamond :: L_< \times L_< \in \{>, <, =\}$;*
- *a* cost bound*, $\delta \in L_<$.*

Then the *extended branching schema*, $branch_{\alpha+}/3$, is obtained from the schema $branch_\alpha/3$ by replacing Line 4 in Figure 6.1 with:

```
if fcost(S) ◇ δ then                    (4*)
    δ ← fcost(S);
    push(P, S);
endif;
```

THEOREM 6.26 *(Properties of the $branch_{\alpha+}/3$ schema) Let $C \in \wp_f(\mathcal{C}^X)$, $S \in \mathcal{SS}^X$, $\varepsilon, \alpha \in \Re^+ \cup \{0.0\}$ and $p = \top_{\Re\mathcal{I}}$. Suppose that the procedure $solve_\varepsilon/2$ terminates for all values*[7]*. Then, the following properties are guaranteed:*

1. Termination*: if $\alpha > 0.0$ then the execution of $branch_{\alpha+}(C, S, p)$ terminates;*

[6] Normally $L_<$ would be $\Re$.

[7] Again note that termination of this procedure is always guaranteed if $\varepsilon > 0.0$ -See Theorem 4.67.

2. *If fcost is a constant function with value δ and $\diamond$ is $=$, then all properties shown in Theorem 6.19 hold for the execution of $branch_{\alpha+}(C, S, p)$.*

3. Soundness on optimisation*: If at least one authentic solution with a cost higher than $\bot_{L<}$ (resp. lower than $\top_{L<}$) exists for $C \cup S$, $\alpha = 0.0$, $\diamond$ is $>$ (resp. $<$), $\delta = \bot_{L<}$ (resp. $\top_{L<}$), the stack P is initially empty and the execution of $branch_{\alpha+}(C, S, p)$ terminates with P non-empty, then the element on the top of P is the first authentic solution found that maximises (resp. minimises) the cost function.*

Observe that the search tree for $branch_{\alpha+}(C, S, p)$ is the same as for $branch_{\alpha}(C, S, p)$. In the following we prove independently each property claimed in Theorem 6.26.

PROOF 6.27 *(Property (1). Termination) This proof is as that of Theorem 6.19(1).*
□

PROOF 6.28 *(Property (2)) Observe that if $fcost(S) = \delta$ for all $S \in \mathcal{SS}^X$, then test in Line 4* of the extended schema always holds. It is straightforward to prove, in this case, that the schemas $branch_{\alpha}/3$ and $branch_{\alpha+}/3$ are equivalent so that all properties of the schema $branch_{\alpha}/3$ hold in the schema $branch_{\alpha+}/3$.*
□

PROOF 6.29 *(Property (3). Soundness on optimisation) We prove the case when $\diamond$ and δ are, respectively, $>$ and $\bot_{L<}$. The respective case is proved analogously. As shown in proof of Theorem 6.19(2), for $\alpha = 0.0$, if $R \in Sol_a(C \cup S)$ then there exists in the search tree some path q of length $j \geq 0$, such that $R = S^f_q$ and the tests in Lines 2-3 hold by Definition 6.6. Thus, Line 4* is reached for all $R \in Sol_a(C \cup S)$, and as consequence, the top of P will contain the first authentic solution found that maximises $fcost/1$.*
□

Unfortunately, if $\alpha > 0.0$, we cannot guarantee that the top of the stack contains an authentic solution or even a partial solution for the optimisation problem. However, if the cost function $fcost/1$ is monotonic, solutions can be compared.

THEOREM 6.30 *(Approximate soundness) Suppose that, for $i \in \{1, 2\}$, P_{α_i} is the constraint store stack resulting from the execution of $branch_{\alpha_i+}(C, S, p)$ where $\alpha_i \in \Re^+ \cup \{0.0\}$. Then, if $\alpha_1 < \alpha_2$ the following property hold.*

If P_{α_1} and P_{α_2} are not empty, and $top(P_{\alpha_2})$ is an authentic solution or covers a solution for $C \cup S$, then, if $fcost/1$ is monotonic and $\diamond$ is $<$ (i.e., a minimisation problem),

$$fcost(top(P_{\alpha_1})) \preceq_{L<} fcost(top(P_{\alpha_2})),$$

and, if $fcost/1$ is anti-monotone and $\diamond$ is $>$ (i.e.,a maximisation problem),

$$fcost(top(P_{\alpha_1})) \succeq_{L<} fcost(top(P_{\alpha_2})).$$

PROOF 6.31 *(Property: Approximate soundness) We prove the case when $\diamond$ is $<$. The respective case (i.e., $\diamond$ is $>$) is proved analogously. We show that during the execution of $branch_{\alpha_1+}(C, S, p)$, Line 4* is reached for some $S^f_{q'} \preceq_s top(P_{\alpha_2})$ (where q' is a path of length $m_1 \geq 0$) and thus $fcost(S^f_{q'}) \preceq_{L<} fcost(top(P_{\alpha_2}))$. It follows that either $S^f_{q'} = top(P_{\alpha_1})$ or $S^f_{q'} \neq top(P_{\alpha_1})$ because there is another store $S^f_{q''} = top(P_{\alpha_1})$ such that $fcost(top(P_{\alpha_1})) \preceq_{L<} fcost(S^f_{q'})$. In both cases it follows that effectively $fcost(top(P_{\alpha_1})) \preceq_{L<} fcost(top(P_{\alpha_2}))$.*

Observe that $top(P_{\alpha_2})$ is in P_{α_2} because there exists a path q of length $m \geq 0$ such that $S^f_q = top(P_{\alpha_2})$ and S^f_q was pushed on the stack because Line 4 is reached and the tests in Lines 2 and 3 holds but, for all proper prefixes of q, it does not hold. Thus S^f_q is consistent and either non-divisible or*

$$p_q < \top_{\Re\mathcal{I}} \text{ and } p_q - precision(S^f_q) \leq (\alpha_2, 0).$$

In addition, for all proper prefixes q_1 of q, $S^f_{q_1}$ is divisible and, either

$$p_{q_1} = \top_{\Re\mathcal{I}} \text{ or } p_{q_1} - precision(S^f_{q_1}) > (\alpha_2, 0).$$

Since $\alpha_1 \leq \alpha_2$, we also have that, for all proper prefixes q_1 of q, either

$$p_{q_1} = \top_{\Re\mathcal{I}} \text{ or } p_{q_1} - precision(S^f_{q_1}) > (\alpha_1, 0).$$

Thus, the node with path q is in the search tree for $branch_{\alpha_1+}(C, S, p)$. Now, we have two cases: (1) S^q_f is consistent and non-divisible. As consequence, S^q_f has no children and Line 4 is reached in the execution of $branch_{\alpha_1+}(C, S, p)$. (2) S^q_f is consistent and divisible. By hypothesis S^q_f is an authentic solution R or covers an authentic solution R for $C \cup S$. Then, reasoning as in proof of Theorem 6.19(3) and by (6.7), there must exists some path q', containing the path q, with no children such that $S^f_{q'}$ is consistent and either $R = S^f_{q'}$ or*

$$p_{q'} < \top_{\Re\mathcal{I}} \text{ and } p_{q'} - precision(S^f_{q'}) \leq (\alpha_1, 0).$$

In both cases, Line 4 is reached. Moreover, by repetitive application of Lemmas 6.13(a) and 6.16(a) it is straightforward to prove that $S^f_{q'} \preceq_s S^f_q$.*

□

A direct consequence of this theorem is that by using a(n) (anti-)monotone cost function, the lower α is, the better the (probable) solution is. Moreover, decreasing α is a means to discard approximate solutions. For instance, in a minimisation problem, if

$$fcost(top(P_{\alpha_1})) \succ_{L<} fcost(top(P_{\alpha_2}))$$

with *fcost*/1 monotonic, then, by the *approximate soundness* property it is deduced that $top(P_{\alpha_2})$ cannot be an authentic solution or cover an authentic solution.

6.5.1 Different Ways to Solve the Instances

In this section, we explain how the choice of the instantiation of the additional global functions and parameters in the definition of $branch_{\alpha+}/3$ determines the method of solving for a set of interval constraints i.e., the schema $branch_{\alpha+}/3$ allows a set of interval constraints to be solved in many different ways, depending on the values for *fcost*, δ and $\diamond$.

Theorem 6.26(2) has shown that to solve classical CSPs, *fcost* should be defined as the constant function[8] δ and the parameter $\diamond$ should have the value $=$. Moreover, Theorem 6.26(3) has shown that a CSP is solved as a COP by instantiating $\diamond$ as either $>$ (for maximisation problems) or $<$ (for minimisation problems). In all cases, the value δ should be instantiated to the initial cost value from which an optimal solution must be found. Some possible instantiations are summarised in Table 6.1 where Column 1 indicates the type of CSP, Column 2 gives any conditions on the cost function, Column 3 gives the range of the cost function (usually, this is $\Re$), Columns 4 gives the initial definition of the $\diamond$ operator, and Columns 5 gives the initial value for δ.

CSP Type	*fcost*	$L_<$	$\diamond$	δ
Classical CSP	constant	$\Re$	$=$	$fcost(S)$
Typical Minimisation COP	any cost function	$\Re$	$<$	$\top_{\Re}$
Typical Maximisation COP	any cost function	$\Re$	$>$	$\bot_{\Re}$
Max-Min COP	any cost function	$\Re \times \Re$	$<$	$\top_{\Re\times\Re}$

Table 6.1: CSP type depends on parameters instantiation

In contrast to typical COPs that usually maintain a fixed criteria (i.e., either maximisation or minimisation of the cost function) and a single lower or upper bound, our schema also permits a mix of the maximisation and minimisation criteria (or even to give priority to some criteria over others). This is the case (see Row 4 of Table 6.1) when $L_<$ is a compound domain and the ordering in $L_<$ determines how the COP will be solved.

EXAMPLE 6.32 *Let $C \in \wp_f(\mathcal{C}^X)$ be a set of interval constraints to be solved as a COP, $L_<$ the domain $\Re^2 = \Re \times \Re$ with ordering*

$$(a, b) < (c, d) \iff (a < c \wedge b \geq d) \vee (a \leq c \wedge b > d),$$

and $fcost :: \mathcal{SS}^X \to L_<$ a cost function on $\Re^2$ defined for any $S \in \mathcal{SS}^X$ as

$$fcost(S) = (fcost_1(S), fcost_2(S))$$

where $fcost_1, fcost_2 :: \mathcal{SS}^X \to \Re$ are cost functions defined on $\Re$. Then, if δ and $\diamond$ are initialised respectively to $<$ and $\top_{\Re^2}$ (as shown in Row 4 of Table 6.1), C is solved by minimising $fcost_1$ and maximising $fcost_2$.

[8] Usually $\delta \in \Re$.

On the other hand, if $<$ is defined lexicographically on $\Re^2$, i.e.,

$$(a,b) < (c,d) \iff a < c \vee a = c \wedge b < d,$$

C is solved by giving priority to the minimisation of $fcost_1$ over the minimisation of $fcost_2$.

For example, suppose $Sol_a(C) = \{S_1, S_2, S_3\}$ and $fcost(S_1) = (1.0, 5.0)$, $fcost(S_2) = (3.0, 1.0)$ and $fcost(S_3) = (1.0, 8.0)$. Suppose also that these solutions have been found by a terminating execution of the $branch_{\alpha+}/3$ schema where $\diamond \equiv <$ and initially $\delta \equiv \top_{\Re^2}$ and that the sequence in which the solutions are found in the search tree is (S_1, S_2, S_3).

Consider the first ordering defined above for $\Re^2$. When S_1 is found, line 4 of the schema is executed with $\delta = (\top_\Re, \bot_\Re)$ (i.e., with $\delta = \top_{\Re^2}$ as shown in Row 4 of Table 6.1) and as consequence S_1 is pushed on the stack P. Afterwards, S_2 is found and line 4* is executed with $\delta = fcost(S_1) = (1.0, 5.0)$. As $fcost(S_2) \not< (1.0, 5.0)$, S_2 is not pushed on the stack. Next S_3 is found and again line 4* is executed with $\delta = fcost(S_1) = (1.0, 5.0)$. As $fcost(S_3) < (1.0, 5.0)$ then S_3 is pushed on the stack so that the top of the new stack contains S_3. Note that S_3 minimises the first component of the cost and maximises the second component.*

Consider next the lexicographic ordering for the domain $\Re^2$. When S_1 is found, line 4 is executed with $\delta = (\top_\Re, \top_\Re)$ (i.e., with $\delta = \top_{\Re^2}$ as shown in Row 4 of Table 6.1) and as consequence S_1 is pushed on the stack P. Afterwards, S_2 is found and line 4* is executed with $\delta = fcost(S_1) = (1.0, 5.0)$. As $fcost(S_2) \not< (1.0, 5.0)$ then S_2 is not pushed on the stack. Finally S_3 is found and again line 4* is executed with $\delta = fcost(S_1) = (1.0, 5.0)$. As $fcost(S_3) \not< (1.0, 5.0)$, S_3 is not pushed on the stack and the top of the stack contains S_1. In this case, S_1 minimises the first component and only if the values of the first components are equal, minimises the second component.*

6.6 A Simple Example

Here we show an example in the integer domain, illustrating the flexibility of the schema to solve a set of interval constraints in different ways. Suppose we want to solve the following set of constraints

$$\begin{aligned} C = \{\ & x_1 + x_2 + x_3 \leq 1, \\ & x_1 \leq 1,\ x_2 \leq 1,\ x_3 \leq 1, \\ & x_1 \geq 0,\ x_2 \geq 0,\ x_3 \geq 0\ \}. \end{aligned}$$

By considering the high level constraints $\leq$ and $plus/3$ as defined in Examples 5.3 and 5.4 (for $L = Integer$) respectively, the set C can be coded in our interval framework

as follows[9]

$$\begin{array}{l}\{\ plus(x_1, x_2, x_{12}),\ plus(x_{12}, x_3, x_{123}),\ x_{123} \leq 1,\\ \quad x_1 \leq 1,\ x_2 \leq 1,\ x_3 \leq 1,\\ \quad 0 \leq x_1,\ 0 \leq x_2,\ 0 \leq x_3\ \}\end{array}$$

where $\mathcal{L} = \{Integer\}$, $X = \{x_1, x_2, x_3, x_{12}, x_{123}\}$ and each variable in X belongs to $V_{Integer}$. Consider also the following cost functions

$$\begin{array}{l} fcost_1, fcost_2 :: \mathcal{SS}^X \rightarrow \Re,\\ fcost_3, fcost_4 :: \mathcal{SS}^X \rightarrow \Re^2,\end{array}$$

defined for each $S = \{x_1 \sqsubseteq r_1,\ x_2 \sqsubseteq r_2,\ x_3 \sqsubseteq r_3,\ x_{12} \sqsubseteq r_{12},\ x_{123} \sqsubseteq r_{123}\}$ in $\mathcal{SS}^X$ as follows

$$\begin{array}{ll} fcost_1(S) = 1.0; & \%\% \text{ Constant function}\\ fcost_2(S) = mid(r_1) + mid(r_2) + mid(r_3); & \%\% \ x_1 + x_2 + x_3\\ fcost_3(S) = (fcost_2(S), mid(r_1) + mid(r_3)); & \%\% \ (x_1 + x_2 + x_3, x_1 + x_3)\\ fcost_4(S) = (fcost_2(S), mid(r_2) + mid(r_3)); & \%\% \ (x_1 + x_2 + x_3, x_2 + x_3)\end{array}$$

where $mid(\{a, b\})$ is a function from $R^s_{\Re}$ to $\Re$ that returns the mid point in the range $\{a, b\}$ i.e., $mid(\{a, b\}) = \frac{a+b}{2.0}$ (e.g., $mid([\mathbf{1.0,4.0}]) = 2.5$).

Consider now the schema $branch_{\alpha+}/3$ with $\alpha = 0.0$ and $\varepsilon = 0.0$, $choose_{naive}$ as defined in Example 6.10 and $split_{Integer}$ as defined in Example 6.12. Now, assume that initially $p = \top_{\Re\mathcal{I}}$, the global stack P is empty and S is the top element of $\mathcal{SS}^X$. Suppose that the schema $branch_{\alpha+}(C, S, p)$ is executed independently with the values for δ, $\diamond$ and $fcost/1$ shown in each of the rows of Table 6.2. Each row is concerning with an execution of the schema. In this table, Columns 2, 3 and 4 shows respectively the value to which δ is initialised, the cost function used and the initialisation of $\diamond$ in the current execution of the schema and

- Column 1 indicates the way in which the CSP is solved (where Max-Min means that we have mixed criterias for the optimisation as done in Example 6.32),
- Column 5 indicates where, in the final state of the stack P, the authentic solution(s) is (are) positioned and

[9]Observe that due to the definition of the plus constraint two intermediate variables x_{12} and x_{123} have been added. However this may be easily avoided by considering alternative definitions e.g., we could declare a plus constraint with four arguments in the following way

$$plus(x, y, z, w) \Leftrightarrow plus(x, y, xy),\ plus(xy, z, w).$$

- Column 6 references the figure that shows the final state[10] of the stack[11] P.

Note that since the integer domain is finite, termination is guaranteed even if $\alpha = 0.0$ and $\varepsilon = 0.0$.

CSP Type	δ	Cost function	$\diamond$	Solution	Figure
Classical CSP	1.0	$fcost_1$	$=$	Any in the stack	6.2(a)
Maximisation COP	$\bot_{\Re}$	$fcost_2$	$>$	stack top	6.2(b)
Minimisation COP	$\top_{\Re}$	$fcost_2$	$<$	stack top	6.2(c)
Max-Min COP (i)	$(\bot_{\Re}, \top_{\Re})$	$fcost_3$	$<_1$	stack top	6.2(d)
Max-Min COP (ii)	$(\bot_{\Re}, \top_{\Re})$	$fcost_4$	$<_1$	stack top	6.2(e)
Max-Min COP (iii)	$(\bot_{\Re}, \top_{\Re})$	$fcost_3$	$<_2$	stack top	6.2(f)
Max-Min COP (iv)	$(\bot_{\Re}, \top_{\Re})$	$fcost_4$	$<_2$	stack top	6.2(g)

Table 6.2: Different solvings of the CSP

Solution S	$fcost_1(S)$	$fcost_2(S)$	$fcost_3(S)$	$fcost_4(S)$
(1,0,0)	1.0	1.0	(1.0,1.0)	(1.0,0.0)
(0,1,0)	1.0	1.0	(1.0,0.0)	(1.0,1.0)
(0,0,1)	1.0	1.0	(1.0,1.0)	(1.0,1.0)
(0,0,0)	1.0	0.0	(0.0,0.0)	(0.0,0.0)

Table 6.3: Evaluation of the solutions to the problems

Each execution of the schema gives rise to a different way of solving $C \cup S$. For instance, Row 1 of Table 6.2 indicates how to solve the problem as a classical CSP. Here *fcost* is a constant function with value δ (where δ is 1.0) and $\diamond$ is $=$. In this case, all authentic solutions are pushed on the stack (see Figure 6.2(a)) as stated in Theorem 6.26(2) (see also Theorem 6.19(2)). Rows 2-3 in Table 6.2 show how the problem can be solved by maximising and minimising the function $fcost_2$ respectively. The optimal solution is that on the top of the stack (see Figures 6.2(b) and 6.2(c)). On their turn, Rows 4-7 indicate how to mix optimisation criterias as done in Example 6.32 where $<_1$ and $<_2$ are defined on $\Re^2$ as follows:

$$(a,b) <_1 (c,d) \iff (a \geq c \wedge b < d) \vee (a > c \wedge b \leq d);$$
$$(a,b) <_2 (c,d) \iff a > c \vee a = c \wedge b < d.$$

Thus, Row 4 corresponds to the problem of maximising $x_1+x_2+x_3$ and minimising x_1+x_3; Row 5 corresponds to maximising $x_1+x_2+x_3$ and minimising x_2+x_3; Row 6

[10] To the right of each element in P we write its cost.

[11] For simplicity (a,b,c) denotes the constraint store

$$S = \{x_1 \sqsubseteq [a,a],\ x_2 \sqsubseteq [b,b],\ x_3 \sqsubseteq [c,c],\ x_{12} \sqsubseteq [d,d],\ x_{123} \sqsubseteq [e,e]\ \}$$

where d and e are any integers - that clearly are not relevant for the solution. For instance, $(0,1,0)$ denotes the constraint store $S = \{x_1 \sqsubseteq [0,0], x_2 \sqsubseteq [1,1], x_3 \sqsubseteq [0,0], \ldots\}$.

Figure 6.2: The final state of the global stack P in the different solvings of the CSP

corresponds to first maximising $x_1 + x_2 + x_3$ and, if this cannot be further optimised, then minimising $x_1 + x_3$ (this is consequence of the ordering $<_2$ that gives priority to the maximisation of the first component over the minimisation of the second one); and Row 7 does the same but minimising $x_2 + x_3$. Figure 6.2 shows the final state of the global stack for each of these cases (in order to help the reader to follow the example, Table 6.3 shows the cost of each authentic solution with respect to the four cost functions considered in this example).

Note that, problems involving other mixed criterias of optimisation may be solved by defining alternative orderings on $\Re^2$.

6.7 Related Work

Constraint solving algorithms have received intense study from many researchers, although the focus has been on developing new and more efficient methods to solve classical CSPs (Freuder and Hubbe, 1995; Wallace, 1993) and partial CSPs (Freuder and Wallace, 1992; Meseguer and Larrosa, 1995). See (Kumar, 1992; Ruttkay, 1998; Smith, 1995; Van Hentenryck, 1995) for more information on constraint solving algorithms and (Kondrak and Van Beek, 1997; Nadel, 1989) for selected comparisons.

Most of the work existing in the literature about the branching step is focused on the discrete domain and in this case branching is usually called *labeling* (Van Hentenryck, 1989). Labeling consists of assigning values (i.e., the instantiation) to the constrained variables and, by a backtracking search, to find a solution (if it exists) for the CSP. The order in which variables and values are instantiated will have a significant influence on the shape of the search tree and thus the performance of the solution (as already discussed in Section 2.2.3).

On infinite domains, labeling is rarely applied as for FD. Of course there are exceptions such as that shown in (Monfroy et al., 1995; Monfroy, 1996) that applied labeling

to process the solutions on infinite and continuous domains. Before labeling is applied, the only values that a variable can take are roots of an univariate polynomial so that in fact only discrete and finite domains are considered.

Traditionally, on the continuous domain (i.e., the real domain) the branching process consists of splitting (usually in two parts) the domain of some variable(s) so as to continue with the search for a solution in each of the derived partitions. This is the process followed in well known systems such as CLP(BNR) (Older and Benhamou, 1993) and CLIP (Hickey, 2000). These systems provide interval constraint solving on which a real variable has associated an interval (in the usual meaning of set theory) and a classical strategy of "divide and conquer" in the solving of problems involving real numbers is usually employed. When no more propagation is possible, the interval solver uses a sort of domain splitting to return each answer. This method is called *split-and-solve* (Benhamou and Older, 1997). The *split-and-solve* method repeatedly selects a variable, splits its associated interval into two or more parts and uses backtracking to look for solutions in each partition. Of course, there is the necessity of a termination test that avoids the infinite splitting of ranges (at least theoretically because in practice the real domain is finite since the precision of a machine is finite). Particularly, CLP(BNR) extends this strategy to the Boolean and integer domains.

Throughout this document we have already shown that our framework allows cooperative instances to solve sets of interval constraints defined on multiple domains. In our framework, the *split-and-solve* method of CLP(BNR) is easily emulated if we consider $\mathcal{L}$ to be the set $\{Integer, \Re, Bool\}$, the definition of the precision maps for the domains in $\mathcal{L}$ of Example 4.62, the definition of the binary splitting functions for the domains in $\mathcal{L}$ of Example 6.12 and the selecting function defined in either Example 6.10 or Example 6.15. To guarantee termination is enough to use the operational procedure $solve_\varepsilon$ with $\varepsilon > 0.0$ (e.g., $\varepsilon = 10^{-6}$) and impose $\alpha > 0.0$.

6.8 Concluding Remarks

In this chapter we have generalised the well known split-and-solve method of the CLP(BNR) system (Benhamou and Older, 1997) to any domain with lattice structure what means that it is valid for both classical domains (i.e., real, integers, Boolean and sets) and new (possibly combined) domains. In this generalisation, we propose an interval branching schema that extends the interval propagation schema described in Section 4.5. This extension gives rise to a generic schema for interval constraint solving that allows problems defined on any set of lattices to be solved in terms of interval constraints.

To achieve this, we have first defined the concept of authentic solution as an assignment of values to variables that satisfies all the constraints. Then, by using a schematic formulation for the branching process, we have indicated which properties of the main procedures involved in branching are responsible for the key properties of interval constraint solving. Then we have extended the schema for optimisation and have shown by means of examples that, in some cases, the methods for solving CSPs

depend on the ordering of the range of the cost functions.

We have also proved key properties such as correctness and completeness and shown how termination may be guaranteed by means of a *precision map* similar to that defined for the propagation schema described in Section 4.5. Moreover, by means of an example, we have also shown how the precision map is a means to normalise the heuristic for variable ordering on systems supporting multiple domains (e.g., cooperative systems).

The schema can be used for most existing constraint domains (finite or continuous) and, as for the propagation framework described in Chapter 4, is totally transparent (i.e., it is a glass box) and also applicable to multiple domains and cooperative systems.

6.9 Contributions

To our knowledge, despite the fact that it is well known that the branching step is a crucial process in complete constraint solving, papers concerned with the general principles of constraint solving algorithms have mainly focused on the propagation step (Apt, 1999; Fernández and Hill, 1999c; Van Hentenryck et al., 1992). The contributions of this chapter are as follows:

- first, to the CLP community, it is an attempt to find general principles for the branching process in interval constraint solving. The branching schema provided here is a generic schema for solving sets of interval constraints on finite and continuous domains as well on multiple domains and it is useful to prove and devise generic properties of interval constraint solving.

  Moreover, the branching schema proposed here defines a generic operational semantics for interval constraint solving that can explain the behaviour of a number of existing interval constraint systems such as such as clp(FD) (Codognet and Diaz, 1996a), clp(B) (Codognet and Diaz, 1994), DecLic (Goualard et al., 1999), clp(B/FD) (Codognet and Diaz, 1996b), CLIP (Hickey, 2000), Conjunto (Gervet, 1997) or CLP(BNR) (Benhamou and Older, 1997);

- secondly, to the thesis, the chapter completes the cooperative and generic schema for interval propagation described in Chapter 4 to interval constraint solving. The resulting framework is a generic, cooperative and transparent setting for interval constraint solving. To our knowledge, with the exception of CHR, no other existing interval constraint system hold all these characteristics so that our interval system is novel.

As minor contributions, this chapter also

- proposes a new method for normalising the heuristic for variable ordering on systems supporting multiple domains. This method is based on the definition of a monotonic precision function on each computation domain that maps "intervals" to values in the totally ordered predefined domain $\Re\mathcal{I}$. As consequence, intervals defined on different domains can be compared via comparing values in this domain;

- shows that, in some cases, the methods for solving CSPs depend on the ordering of the range of the cost functions;

Part IV

Practical Framework

Chapter 7

A 2D Glass Box, Collaborative, Generic CLP Language

The limits of my language
mean the limits of my world.

Tractatus Logico-Philosophicus (1922)
Ludwig Wittgenstein 1889-1951

7.1 Motivation

In order to validate the feasibility of our ideas, we have implemented a language for interval constraint solving on lattices based on the theoretical framework described in Part III.

In this chapter we propose a new CLP language, called $clp(\mathcal{L})$ (i.e., CLP on sets of lattices), that is totally based in the theoretical framework shown in previous chapters. The $clp(\mathcal{L})$ language is an alternative for a flexible constraint solver that allows for user and system defined constraints (that is a *glass box on constraints* approach) as well as user and system defined domains (that is a *glass box on domains* approach) with lattice structure, even if the cardinality is infinite, and interaction between them (i.e., *cooperation*). As $clp(\mathcal{L})$ provides a 2-dimensional (2D) glass box approach on constraints and on domains (see Figure 1.3 on page 7) we say that $clp(\mathcal{L})$ is *transparent*.

Here we show the main features of $clp(\mathcal{L})$, discuss briefly a prototype implementation and show several examples of its flexibility. Observe that we do not pretend to give the whole specification of both the language and the prototype implementation, since this can be found in (Fernández, 2000), but only a global overview.

7.1.1 Chapter Structure

This chapter is structured as follows. In Section 7.2 we introduce the basic syntax of the $clp(\mathcal{L})$ language and show several examples of how to declare both user-defined domains and constraints. Then Section 7.3 describes the execution procedure of the $clp(\mathcal{L})$ system (i.e., our prototype implementation of the $clp(\mathcal{L})$ language) by showing how the unification step of usual LD-resolution is extended. Section 7.4 describes how the $clp(\mathcal{L})$ system is implemented and enumerates the functionalities available in the current version. In Section 7.5, three $clp(\mathcal{L})$ programs are developed to show the flexibility of the system. The chapter ends with a discussion about related work, the conclusions and a summary of the main contributions of the chapter.

7.2 The $clp(\mathcal{L})$ language

The $clp(\mathcal{L})$ language combines the features of standard Prolog (Sterling and Shapiro, 1986) with some extra declarations such as: (1) declarations of the computation domains (including the combined domains) and (2) declarations of the constraint operators on the computation domains.

7.2.1 Domain Declarations

A *domain declaration* allows the user to define a new domain with a lattice structure. Basically, the predicates *lattice/2, finite/1, lt/4, glb/5 and lub/5* are used to declare a new lattice as computation domain. If the lattice is totally ordered, then the predicates *succ/3 and pred/3* can also be used.

- **lattice(D,E)** declares that the element **E** belongs to the domain **D**.
- **finite(D)** is true if **D** is finite and false otherwise.
- **lt(normal,D,X,Y)** declares that **X** is lower than **Y** in the domain **D**.
- **glb(normal,D,X,Y,Z)** defines **Z** to be the greatest lower bound (glb) of the elements **X** and **Y** in the domain **D**.
- **lub(normal,D,X,Y,Z)** defines **Z** to be the least upper bound (lub) of the elements **X** and **Y** in the domain **D**.
- **succ(D,X,Y)** defines **Y** to be the immediate successor of the element **X** in the totally ordered domain **D**. It must be defined for all element **X** in **D**.
- **pred(D,X,Y)** defines **Y** to be the immediate predecessor of the element **X** in the totally ordered domain **D**. It must be defined for all element **X** in **D**.

The first argument in the predicates *lt/4, glb/5 and lub/5* is instantiated to 'normal' that is a predefined constant term used to distinguish the user-defined declarations from other built-in declarations.

New domains resulting from the combination of existing (user or system) domains can also be defined by using the predicates *product_Direct/3*, *product_Lexicographic/3* and *linear_sum/3*.

- **product_Direct(D,E,F)** declares **F** to be the direct product of **D** and **E**.
- **product_Lexicographic(D,E,F)** declares **F** to be the lexicographic product of **D** and **E**.
- **linear_sum(D,E,F)** declares **F** to be the linear sum of **D** and **E**.

EXAMPLE 7.1 *The real domain can be declared as follows:*

```
lattice(real,Ele):-float(Ele).
lt(normal,real,A,B):- lattice(A,real),lattice(B,real),!,A < B.
finite(real):-fail.
glb(normal,real,X,Y,Z):- lattice(real,X),lattice(real,Y),
                         (X<=Y->Z=X;Z=Y).
lub(normal,real,X,Y,Z):- lattice(real,X),lattice(real,Y),
                         (X>Y->Z=X;Z=Y).
```

float/1 *is a built-in predicate in clp($\mathcal{L}$) (see Section 7.4). A call* float(X) *is true if X is a real value.*

The following clause declares the integer point domain to be the direct product $\langle Integer, Integer\rangle$ (where Integer is declared analogously to the real domain -see Appendix A).

```
product_Direct(integer,integer,int_point).
```

More examples of declarations of new computation domains in the *clp($\mathcal{L}$)* language are shown in Appendix A.

7.2.2 Declarations of Constraint Operators.

The current implementation of *clp($\mathcal{L}$)* allows both unary and binary operators (see Section 4.3.2 on page 88). As usual in a logic language, all the constraint operators have to be declared as dynamic[1]. The declaration of a constraint operator also requires the use of the predicate *declara/3* or *declara/4*. Let L, L_1, L_2 be (user or system) (not necessarily distinct) computation domains.

- **declara(Op,$\mathbf{L_1}$,L)** specifies the unary operator $Op :: L_1^s \rightarrow L^s$.
- **declara(Op,$\mathbf{L_1}$,$\mathbf{L_2}$,L)** specifies the binary operator $Op :: L_1^s \times L_2^s \rightarrow L^s$.

[1]This is just a matter of the current implementation prototype.

If the $\mathbf{L}_1$ or $\mathbf{L}_2$ is replaced in the above by **mirror($\mathbf{L}_1$)** or **mirror($\mathbf{L}_2$)**, then the domains L_1^s or L_2^s are replaced by the mirrors $\overline{L_1^s}$ or $\overline{L_2^s}$.

Also, any declared operator has to be defined on both the bracket domain and the computation domain (see Definition 4.10 on page 88). From these definitions the $clp(\mathcal{L})$ system infers the definition of the constraint operator (by (4.2) in Page 88) as well as the definition of its mirror operator (by (4.3) in Page 88). As a consequence, if Op is a (unary or binary) constraint operator defined on both the bracket domain and a computation domain L, $t_i \in L_i^s$ and $t_i = a_i\mathbf{\}}_i$ or $t_i \in \overline{L_i^s}$ and $t_i = \mathbf{\{}_i a_i$, for $i \in \{1,2\}$, then the definition of the constraint operator over the domain L^s is inferred as follows:

$$\begin{aligned} &Op(t_1) = Op(a_1)\ Op(\mathbf{\}}_1) && \text{for unary operators} \\ &Op(t_1,t_2) = Op(a_1,a_2)\ Op(\mathbf{\}}_1,\mathbf{\}}_2) && \text{for binary operators.} \end{aligned} \tag{7.1}$$

EXAMPLE 7.2 *Assume that the integer and real domains have already been declared and that the integer point domain is as in Example 7.1 on the preceding page (see their complete declaration in Appendix A). The following $clp(\mathcal{L})$ clauses declare the constraint operators $+$ and $-$ as defined in Example 4.11 on page 88. Observe that these operators have to be defined on the bracket as well as on the computation domains*[2].

```
    %--- Declaring the operators dynamically as usual
:-dynamic ':+:'/3.
:-op(625,xfx,':+:').

:-dynamic ':-:'/3.
:-op(625,xfx,':-:').

    %--- Operator declarations
declara(:+:,L,L,L).                                              (1)
declara(:-:,L,mirror(L),L).

    %--- Definition on the bracket domain.
:+:(close,close,close).                                          (2)
:+:(open,_,open).
:+:(_,open,open).

:-:(A,A,close).                                                  (3)
:-:(A,B,open):-A\==B.

    %--- Definition on the integer, real and integer point domains
:+:(E1,E2,E3):-(lattice(integer,E1), lattice(integer,E2));       (4)
            (lattice(real,E1), lattice(real,E2)),E3 is  E1+E2.
:+:((A,B),(C,D),(E,F)):-lattice((A,B),int_point),
```

[2]By syntax conventions usually a constraint operator in the current implementation of the $clp(\mathcal{L})$ system begins and ends with colon.

```
                        lattice((C,D),int_point),
                        E is  A + C, F is B + D.
    %--- Definition on the integer, real and integer point domains
:-:(E1,E2,E3):-(lattice(integer,E1), lattice(integer,E2));       (5)
             (lattice(real,E1), lattice(real,E2)),E3 is  E1-E2.
:-:((A,B),(C,D),(E,F)):-lattice((A,B),int_point),
                        lattice((C,D),int_point),
                        E is  A - C, F is B - D.
```

In clp($\mathcal{L}$), 'open' and 'close' are reserved words denoting the brackets) and] respectively. Note that the operators :+: and :−: are declared in line (1) generically for any L as

$$\text{`}:+:\text{'} :: L^s \times L^s \to L^s$$
$$\text{`}:-:\text{'} :: L^s \times \overline{L^s} \to L^s$$

and defined respectively in lines (4) and (5) for the real, integer and integer point domains. In lines (2) and (3) these operators are defined on the bracket domain as shown in Example 7.1 on page 175, that is to say, for each $\}_1, \}_2 \in B$

$$\}_1 :+: \}_2 = min_B(\}_1, \}_2),$$
$$\}_1 :-: \}_2 = \text{] } \textit{iff } \}_1 = \}_2 \textit{ and } \text{) } \textit{otherwise.}$$

From (7.1), the system infers the global definition for :+: and :−: in L^s (where L is the integer domain, the real domain or the integer point domain) and from (4.3) *on Page 88 deduces its definition in $\overline{L^s}$.*

It is also important to mention that the *clp*($\mathcal{L}$) system infers, from the declarations of the constraint operators, the validity of the interval constraints with respect to the theoretical framework. This guarantees the monotonicity property defined in Section 4.4.6 on page 104.

More examples of declarations of constraint operators in the *clp*($\mathcal{L}$) language are shown in Appendix B.

7.2.3 Interval Constraints

An interval constraint $x \sqsubseteq r$ is expressed in *clp*($\mathcal{L}$) by an expression of the form

X in R

where X is a logical variable and R a range in *clp*($\mathcal{L}$).

The basic syntax of the X *in* R constraint in *clp*($\mathcal{L}$) is shown in Table 7.1 where: 'bottom' and 'top' are reserved words denoting, respectively, fictitious bottom and top elements of any computation domain; 'open' and 'close', as already said, are also reserved words denoting the brackets) and] respectively; d is any value belonging

Table 7.1: Basic syntax of the constraint X *in* R in $clp(\mathcal{L})$

interval constraint ::=	X *in* range	(interval constraint)
range::=	term..term	(range)
term::=	'('constant','bracket')'	(term in L^s or $\overline{L^s}$ for some $L \in \mathcal{L}$)
	'('*min Y*')'	(minimum indexical term)
	'('*max Y*')'	(maximum indexical term)
	'('*val Y*')'	(ground indexical term)
	δ(term)	(δ is a unary operator for L^s)
	'('term $\circ$ term')'	($\circ$ is a binary operator for L^s)
bracket::=	open	(open bracket ')')
	close	(close bracket ']')
constant::=	d	($d \in L$ for some $L \in \mathcal{L}$)
	bottom	(lifted bottom element)
	top	(lifted top element)

to some (user or system defined) domain L already declared as in Section 7.2.1; $\circ$ is a binary operator[3] (for L^s) and δ is a unary operator (for L^s) already declared as in Section 7.2.2; X and Y are logical variables ; a term '$(d, close)$' represents either the term $\mathbf{d}] \in L^s$ or the term $[\mathbf{d} \in \overline{L^s}$ whereas a term '$(d, open)$' represents either the term $\mathbf{d}) \in L^s$ or the term $(\mathbf{d} \in \overline{L^s}$.

EXAMPLE 7.3 *Suppose that operators :+: and :-: are declared in the $clp(\mathcal{L})$ system as shown in Example 7.2. Then, in $clp(\mathcal{L})$*

'X in (4,close)..(9,open)' denotes the constraint $x \sqsubseteq [4, 9)$.

'X in ((4,close):+:(min Y))..(max Y)' denotes $x \sqsubseteq [4 \overline{+} min(y), max(y)$.

'X in ((0,0),close)..((max Z):+:((3,1),close))' denotes $x \sqsubseteq [(0, 0), max(z) + (3, 1)]$.

where 4 and 9 are integers, and (0,0) and (3,1) are elements of the domain 'int_point' (see Example 7.1).

Observe that the position of a constraint operator (together with its declaration) determines if we are using the operator or its mirror.

7.2.4 High Level Constraints

From the interval constraints in $clp(\mathcal{L})$, it is possible to construct high level constraints.

[3]Observe that infix notation is allowed for the binary operators.

EXAMPLE 7.4 *Consider the operators :+: and :-: as declared in Example 7.2. The following clp($\mathcal{L}$) code defines the (overloaded) high level constraint* plus/3 *as shown in Section 5.4 on page 127.*

```
plus(X,Y,Z) :- X in ((min Z):-:(max Y))..((max Z):-:(min Y)),
                Y in ((min Z):-:(max X))..((max Z):-:(min X)),
                Z in ((min X):+:(min Y))..((max X):+:(max Y)).
```

Observe that this constraint can generically be applied on all computation domains in which operators :+: and :-: have been defined. For example, consider the domains of integer points and reals as declared in Example 7.1 as well as the domains of integers and sets (with no specific type base) that are declared in Appendix A (the integer domain is declared as usual and the set domain is defined as a list with the inclusion as ordering). Consider also the definition of the constraints operators :+: and :-: for the integer, real and integer point domains as shown in Example 7.2. In the domain of sets, the operators :+: and :-: are defined to be the usual union and difference of sets respectively (see Appendix A on page 227). Then, a goal such as

```
[X,Y,Z]::'real,   [V,W,T]::'integer,
[C1,C2,C3]::'set, [P1,P2,P3]::'int_point,
Z in (1.0,close)..(4.0,close), Y in (0.0,open)..(90.0,close),
V in (1,close)..(2,close), W in (2,close)..(9,open),
C1 in ([1],close)..([1,2,3],close), C2 in ([4],close)..([4,7],close),
P1 in ((0,0),close)..((1,2),close), P2 in ((1,3),close)..((2,9),close),
plus(X,Y,Z),
plus(V,W,T),
plus(C1,C2,C3),
plus(P1,P2,P3).
```

returns the solution

```
X in (-89.0,close)..(4.0,open),
T in (3,close)..(10,close),
C3 in ([1,4],close)..([1,2,3,4,7],close),
P3 in ((1,3),close)..((3,11),close).
```

In clp($\mathcal{L}$), x::'L denotes a type constraint for x in L (see Definition 4.26 on page 96) and $[x_1,\ldots,x_n]$::'L is equivalent to x_1::'L, ..., x_n::'L. Thus, X, Y and Z are constrained in the real domain, V, W and T in the integer domain, C1, C2 and C3 in the domain "sets of integers" and P1, P2 and P3 in the domain of integer points. Observe that the constraint 'T in (3,close)..(11,open)' was reduced to the constraint 'T in (3,close)..(10,close)' by applying of the equivalence rules for discrete domains (see Section 4.4.4 on page 102).

Note again that the high level constraint plus/3 is used generically on very different domains with no special reasoning for each domain.

EXAMPLE 7.5 *The generic high level constraint $\leq$ defined in Example 5.3 on page 127 is coded in the clp($\mathcal{L}$) language as follows:*

```
X <=: Y :- X in (bottom,close)..(max Y),
           Y in (min X)..(top,close).
```

Observe that this constraint is generic since any domain has (lifted) top and bottom elements.

More examples of definitions of high level constraints in the *clp*($\mathcal{L}$) language are shown in Appendix C.

Our prototype implementation also provides an interpreter of the *clp*($\mathcal{L}$) language that allows a syntactic sugar in the command line. For example, the goal shown in Example 7.4 can be directly coded as[4]:

```
clp(L) >  [X,Y,Z]::'real, [V,W,T]::'integer,
          [C1,C2,C3]::'set, [P1,P2,P3]::'int_point,
          Z in [1.0,4.0], Y in (0.0,90.0],
          V in [1,2], W in [2,9),
          C1 in [{1},{1,2,3}], C2 in [{4},{4,7}],
          P1 in [#0,0#,#1,2#], P2 in [#1,3#,#2,9#],
          plus(X,Y,Z),
          plus(V,W,T),
          plus(C1,C2,C3),
          plus(P1,P2,P3).
```

7.3 The Execution Procedure

The *clp*($\mathcal{L}$) system manages a single constraint store which contains all the (user or system defined) constraints for any computation domain. As in Prolog, the resolution mechanism uses LD-resolution but with the unification step extended as explained in the following.

Assume that the system supports n (user or system defined) computation domains and that the computation domains are denoted by L_i $(1 \leq i \leq n)$. Assume also that there are n disjoint sets of variables, denoted by V_{L_i} $(1 \leq i \leq n)$ and that V_{L_i} is associated with the computation domain L_i. A constrained variable $x \in V_{L_i}$ $(1 \leq i \leq n)$ can be unified with

- An unbound variable y: in this case y is just bound to x.
- An element $l \in L_i$: this is equivalent to adding $x \sqsubseteq [l, l]$ to the constraint store.
- Another constrained variable $y \in V_{L_i}$: this is equivalent to adding $x \sqsubseteq min(y), max(y)$ and $y \sqsubseteq min(x), max(x)$ to the constraint store.

[4] 'clp(L) >' is the prompt of the interpreter prototype for the *clp*($\mathcal{L}$) language -see (Fernández, 2000) for more information.

The prototype implementation of the $clp(\mathcal{L})$ language returns a fail, called *domain fail*, if

- x tries to unify with another constrained variable $y \in V_{L_j}$ such that $j \neq i$,
- x tries to unify with an element l and $l \notin L_i$ or
- x is constrained (by a simple interval constraint) to have values in a domain $R_{L_j^b}$ distinct from $R_{L_i^b}$.

EXAMPLE 7.6 *Let "p(X) :- X in [3,5]." be the only clause for p/1 in a $clp(\mathcal{L})$ program where $X \in V_{Integer}$. Consider the following $clp(\mathcal{L})$ goals in the command line:*

```
(a) clp(L) > p(Y).
(b) clp(L) > p(4).
(c) clp(L) > Y in [4,14), p(Y).
(d) clp(L) > Y in [8.9,14.2), p(Y).
(e) clp(L) > p(2.1).
```

Then, assuming Y is an unbound variable, the system:

(a) unifies X with Y and adds the constraint "Y in [3,5]" to the store;

(b) unifies X with 4 and adds the constraint "X in [4,4]" to the store. Then the constraint "X in [3,5]" is also added and the propagation algorithm shown in Section 4.5.1 on page 108 is applied. This case is equivalent to unifying X with 4 and adding "4 in [3,5]" to the store which operationally is just equivalent to testing that $3 \leq 4 \leq 5$ (but this is an improvement to the implementation level);

(c) adds the constraint "Y in [4,14)" to the store and then unifies X with Y (since $Y \in V_{Integer}$) adding also the constraints "X in min(Y),max(Y)" and "Y in min(X),max(X)". Finally, the constraint "X in [3,5]" is added and constraint propagation is executed by following the operational schema described in Section 4.5.1.

(d) adds the constraint "Y in [8.9,14.2)" to the store. Since this implies that $Y \in V_{\Re}$ while $X \in V_{Integer}$, Y and X fail to unify and a domain fail is returned;

(e) tries to unify X with 2.1 and a domain fail is returned since $X \in V_{Integer}$ and $2.1 \notin Integer$.

7.4 The Current $clp(\mathcal{L})$ Implementation

In this subsection, we briefly describe our prototype implementation of $clp(\mathcal{L})$ (Fernández, 2000). This has been built on the SICStus 3#7 Prolog platform (Carlsson et al., 1997).

7.4.1 Interval Constraints

Constraint consistency, store stabilisation and constraint propagation are implemented using the constraint handling rules (CHRs) (Frühwirth, 1998) that are part of a SICStus library. The CHRs are very appropriate since they are solved prior to the resolution step of the standard logical engine. As shown in Section 2.4.2 on page 28 there are three kinds of CHRs, the simplification rule, the propagation rule and the simpagation rule. In the following we briefly detail some aspects of our prototype implementation.

The inconsistency of simple interval constraints (see Definition 4.21 on page 93) is defined via the following propagation CHRs[5]:

$$inconsistency_1 \ @ \ x \sqsubseteq \overline{s}, t \Rightarrow simple(\overline{s}, t), not(s \leq_L t) \ \mid fail.$$
$$inconsistency_2 \ @ \ x \sqsubseteq (a, a\} \Rightarrow fail.$$

The test $simple(r)$ is a built-in predicate that returns true if and only if the range r is simple (see definition 4.26) and false otherwise.

Store stabilisation (see Definition 4.31 on page 99) is mainly based on the rule $\cap_L$ (see Definition 4.28 on page 98) that is basically implemented via a simplification CHR as follows:

$$x \sqsubseteq r_1, x \sqsubseteq r_2 \Leftrightarrow simple(r_1), \ simple(r_2), \ glb_{R_L}(r_1, r_2, r_3) \ \mid \ x \sqsubseteq r_3.$$

In a previous step to store stabilisation, for any simple constraint $x \sqsubseteq r$, the system deduces the computation domain L over which the range r is constructed. This means that the system infers the computation domains L_1 and L_2 for r_1 and r_2 respectively and checks that L_1 and L_2 coincide. $glb_{R_L}/3$ is a built-in predicate that given two simple ranges constructed over the same interval lattice R_L^s computes their greatest lower bound. Thus, $glb_{R_L}(r_1, r_2, r_3)$ returns in r_3 the glb in the interval lattice R_L^s of r_1 and r_2. The test $simple$ on r_1 and r_2 checks that r_1 and r_2 are simple (as required in the definition of rule $\cap_L$).

Constraint Propagation (see Section 4.4.3 on page 100) is implemented via a set of propagation CHRs that cover all the possible cases of evaluation of indexicals terms (see Definition 4.39 on page 101). For example, the following propagation CHR

$$x \sqsubseteq (min \ y), t, \quad y \sqsubseteq \overline{s_1}, t_1 \Rightarrow (x \neq y, simple(\overline{s_1}, t_1)) \ \mid \ x \sqsubseteq \overline{s_1}, t$$

is the implementation of the case in which the term $min(y)$ is propagated in the way

$$x \sqsubseteq (min \ y), t \rightsquigarrow^S x \sqsubseteq \overline{s_1}, t$$

where S is a simple stable store containing the simple interval constraint $y \sqsubseteq \overline{s_1}, t_1$ (see Section 4.4.3 on page 100).

[5]The syntax of the CHRs has been simplified in favour of clarity.

7.4.2 Current Resources of the Prototype Implementation

The current clp($\mathcal{L}$) implementation supports user-defined domains (see Section 7.2.1) as well as the predefined domains Boolean, integer, real, natural, colors (a finite set of colors), sets and pairs of integers and reals. As implemented on top of the SICStus system, the prototype also supports the built-in predicates of this system.

It allows user-defined (unary or binary) constraint operators (see Section 7.2.2) and provides predefined Boolean constraints such as *and*/3, *or*/3, *xor*/3, *equiv*/3 and *not*/2 among others; symbolic constraints such as *at_least_one*/1, *at_most_one*/1 and *only_one*/1; arithmetic constraints such as *plus*/3, *diff*/3, *divide*/3 and *times*/3 as well as linear arithmetic constraints such as $=/2, \neq/2, >/2, \geq/2, </2$ and $\leq/2$ defined on usual numerical domains and on combined domains (see Appendix C).

As well, it provides a system-defined predicate to execute a naive labeling strategy[6] on finite lattices in which the successor and predecessor of each element is defined via the corresponding predicates succ/3 and pred/3 described in Section 7.2.1. For example, a naive strategy for labeling in the finite domain is implemented by means of one simplification CHR and two simpagation CHRs that are shown below.

```
labeling([]):-!.
labeling([X|Xs]):-label(X),labeling(Xs).

constraints label/1.
labeling0 @  label(X) <=> ground(X) | true.
labeling1 @  X in (S,close)..(S,close) \ label(X) <=>
                            (lattice(S,L)) | X=S.
labeling2 @  label(X) \ X in (Ele,close)..T1 <=>
             (constant(T1),          %% T1 contains no indexical terms
              lattice(Ele,L),
              finite(L),
              succ(L,Ele,Ele1))
                | ( (X=Ele)         %% Instantiation step
                    ;
                    (X in (Ele1,close)..T1
                    )
                  ).
```

The precision map (see Section 4.5.2) can also be defined by means of a (dynamic) predicate *precision* (see example in Appendix A on Page 230).

Appendices A-C show a significant number of examples. We do not pretend in this document to give a detailed description of the system but only a global overview. For further details about the implementation and more examples see (Fernández, 2000).

[6]Possible values are assigned, by enumeration, to the constrained variables and the constraints checked for consistency. See Section 2.2.3.

7.5 Programming with $clp(\mathcal{L})$

In this section we develop three examples of $clp(\mathcal{L})$ programs in order to highlight the flexibility of the language. The first example, in Section 7.5.1, provides an idea about the generic power of the $clp(\mathcal{L})$ system by showing how the same formulation of a problem leads to different instances solved on different computation domains. In Section 7.5.2, a simple geometry example is used to show how the $clp(\mathcal{L})$ system enables the generation of one-way channels on which the information can flow from one domain to another one. Finally, Section 7.5.3 presents the $clp(\mathcal{L})$ solution to the problem of diagnosing the functionality of the heart proposed in Section 5.5.2 on page 137.

These examples give an idea of the cooperative mechanism and illustrate the flexibility of the system. More examples are shown in Appendix D.

7.5.1 A Generic Scheduling Problem

Consider the simple scheduling problem [7] where there are a number of tasks represented by terms *Task(S,D)* where S is the start time of a task and *D* is the duration of the task. The predicate *in(Task,SuperTask)* is true if the interval for *SuperTask* contains the interval for Task. *noOverlap(Task,Tasks)* is true if *Task* overlaps with none of the tasks in the list *Tasks*. The formulation of this predicate uses two high level constraints (i.e., *plus*/3 and <=:/2) to make sure *Task* is either before or after all the tasks in *Tasks*. The predicate *Schedule(Tasks, Supertask)* is true if all the tasks in the list *Tasks* are in *SuperTask* but no pair in *Tasks* overlap. The $clp(\mathcal{L})$ program to solve this problem is shown below:

```
into(task(S1,D1),task(S2,D2)):- S2 <=: S1, plus(S1,D1,SD1),
                                plus(S2,D2,SD2), SD1 <=: SD2.
noOverlap(_,[]).
noOverlap(task(S1,D1),[task(S2,D2)|Tasks]):-
               ((plus(S1,D1,SD1), SD1 <=: S2)
               ;
                (plus(S2,D2,SD2), SD2 <=: S1)),
               noOverlap(task(S1,D1),Tasks).
schedule([],_).
schedule([Task|Tasks],Supertask):- into(Task,Supertask),
               noOverlap(Task,Tasks),
               schedule(Tasks,Supertask).
```

This program makes use of the high level constraints *plus*/3 and <=:/2 that were coded in Examples 7.4 and 7.5 respectively.

[7] This example is a modification of a program proposed in (Sidebottom and Havens, 1992) to schedule tasks, just considering the real domain, using some of the relations on temporal intervals described in (Allen, 1983).

Observe that no explicit domain is specified since the genericity of these constraints assures that the problem can be solved on different domains with no special reasoning for them. In the following we show different instances.

An instance in CLP(FD). Below, it is shown how queries in the FD can be solved from the command line of our prototype.

```
clp(L) > schedule([task(0,1),task(3,1),task(S,1)],task(0,6)).
     yes.
     The store contains the following constraints
       S in [4,5]
       S in (min _24271):-:[1,(max _24271):-:1] ? |: ;
     yes.
     The store contains the following constraints
       S in (min _26743):-:[1,(max _26743):-:1]
       S in [1,2]
       S in (min _27757):-:[1,(max _27757):-:1] ? |: ;
     no.
```

This query returns the solution "S in [1,2] $\cup$ [4,5]". Figure 7.1(a) shows the graphical representation of the goal. Black points marks the solution set.

A classical instance in CLP($\Re$). Below we show how the problem can be translated to the continuous domain.

```
clp(L) > schedule([task(0.0,0.7),task(2.75,1.0),task(S,0.875)],
                                                  task(0.0,6.0)).
     yes.
     The store contains the following constraints
       S in [3.75,5.125]
       S in (min _24433):-:[0.875,(max _24433):-:0.875] ? |: ;
     yes.
     The store contains the following constraints
       S in (min _26897):-:[0.875,(max _26897):-:0.875]
       S in [0.7,1.875]
       S in (min _27943):-:[0.875,(max _27943):-:0.875] ? |: ;
     no.
```

This query returns the solution "S in [0.7,1.875] $\cup$ [3.75,5.125]". Figure 7.1(b) shows the graphical representation of the goal. Now the black lines mark the solution sets.

A mixed instance in CLP($\Re\times$ FD). A more interesting case is obtained by mixing the preceding instances. Suppose there are two processes *p1 = (S1,D1)* and *p2 = (S2,D2)* where *p1* must be executed on a machine A in real time and *p2* on a machine

B in discrete time. Then a task consists in the resolution of both processes and can be represented as the term *Task((S1,S2),(D1,D2))*. We first add, to the above program, the clause "product_Direct(real,integer,reint_point)" to declare the domain *reint_point* to be the direct product $\Re \times$ *Integer*. Then we extend the definition of :+: and :-: shown in Example 7.2 to this domain by defining $(a,b)+(c,d)=(a+c,b+d)$ and $(a,b)-(c,d)=(a-c,b-d)$. The *clp*($\mathcal{L}$) code is shown below.

```
    %-----  Domain declaration
product_Direct(real,integer,reint_point)                          (1)
    %----- Definition of + and - in the point domain
:+:((E11,E12),(E21,E22),(E31,E32)):-lattice(reint_point,(E11,E12)),
                                lattice(reint_point,(E21,E22)),
                                E31 is E11+E21, E32 is E12+E22.   (2)
:-:((E11,E12),(E21,E22),(E31,E32)):-lattice(reint_point,(E11,E12)),
                                lattice(reint_point,(E21,E22)),
                                E31 is E11-E21, E32 is E12-E22.   (3)
```

Line 1 is used to declare the domain *reint_point* whereas Lines 2 and 3 extend the definition of operators :+: and :-: shown in Example 7.2 to the domain *reint_point*. As direct consequence, the *plus* constraint defined in Example 7.4 can now be applied on variables constrained in this domain and thus also the *schedule* predicate.

Below we show an example of resolution from the command line of our *clp*($\mathcal{L}$) interpreter.

```
clp(L)>schedule([task((0.0,0),(0.7,1)),task((2.75,3),(1.0,1)),
                      task(S,(0.875,1))],task((0.0,0),(6.0,6))).
    yes.
    The store contains the following constraints
      S in [(3.75,4),(5.125,5)]
      S in (min _28704):-:[(0.875,1),(max _28704):-:(0.875,1)] ? |: ;
    yes.
    The store contains the following constraints
      S in (min _31376):-:[(0.875,1),(max _31376):-:(0.875,1)]
      S in [(0.7,1),(1.875,2)]
      S in (min _32582):-:[(0.875,1),(max _32582):-:(0.875,1)] ? |: ;
    no.
```

This query returns the solution "S in [(3.75,4),(5.125,5)] $\cup$ [(0.7,1),(1.875,2)]" that is graphically illustrated in Figure 7.1(c). Again black lines mark the solution set

The solution is interpreted as follows: process *p1* has to begin its execution in machine A during the interval $[3.75, 5.125]$ and, in this case, process *p2* has to start its execution, in machine B, during the interval $[4, 5]$ (i.e., in the fourth or fifth unit of time in machine B). Alternatively, *p1* can begin its execution during the interval $[0.7, 1.875]$ and, then, *p2* has to start during the interval $[1, 2]$ (i.e., in the first or second unit of time in machine B).

Figure 7.1: Solving a scheduling problem

7.5.2 A Geometry Problem Solved by Solver Collaboration

The $clp(\mathcal{L})$ language also allows information flow between different (possibly combined) computation domains. Consider the simple problem of computing the pairs of reals $(x_1, y_1), (x_2, y_2)$ such that

$$x_1 + y_1 = x_2 - y_2 = Cons1$$

in a plane $(0.0, 0.0) \times (Cons2, Cons2)$ (where $Cons1, Cons2 \in \Re$).

This problem could be solved in the real domain. However, to show the flexibility of the system we will solve it in such a way that the solution in the real domain is propagated to the real point domain.

Let the following $clp(\mathcal{L})$ program:

```
    %----- Declaration of domain
product_Direct(real,real,real_point)                          (1)
    %----- Declaration of operator
:-dynamic ':@:'/3.
:-op(625,xfx,':@:').
declara(:@:,real,real,real_point).                            (2)
    %----- Operator definition on the bracket domain
:@:(open,_,open).    :@:(close,B,B).                          (3)
    %----- Operator definition on the real_point domain
:@:(A,B,(A,B)):-lattice(real,A),lattice(real,B).              (4)
```

In this program, the domain *real_point* is declared as the direct product $\langle \Re, \Re \rangle$ (line 1) whereas the operator :@: is defined as follows

$$\text{`}:@:\text{'} :: \Re^s \times \Re^s \rightarrow \mathit{real\_point}^s$$
$$\mathbf{a}\}_1 :@: \mathbf{b}\}_2 = \mathbf{a}: @ :\mathbf{b}\ \}_1 : @ :\}_2 = (\mathbf{a}, \mathbf{b})\ \mathit{min}_B(\}_1, \}_2)$$

where $\mathbf{a}\}_1, \mathbf{b}\}_1 \in \Re^s$. For instance, **1.5**] : @ : **2.0**] = **(1.5,2.0)**]. Observe that this operator is used to combine elements (and thus propagates values from the real domain to the domain of real points).

Suppose *Cons1* = 3.5 and *Cons2* = 20.0. Below we show the query that is typed from the command line of our $clp(\mathcal{L})$ interpreter to solve this problem[8].

```
clp(L) >  [X1,Y1,X2,Y2] in [0.0,20.0],
     [P1,P2] in [#0.0,0.0#,#20.0,20.0#],
     P1 in ((min X1):@:(min Y1)),((max X1):@:(max Y1)),
     P2 in ((min X2):@:(min Y2)),((max X2):@:(max Y2)),
     plus(X1,Y1,3.5),plus(Y2,3.5,X2).
 yes.
 The store contains the following positive constraints
   P2 in [(3.5,0.0),(20.0,16.5)]
   P2 in (min _29855):@:(min _29888),(max _29855):@:(max _29888)
   P1 in [(0.0,0.0),(3.5,3.5)]
   P1 in (min _31733):@:(min _31766),(max _31733):@:(max _31766) ? |: ;
 no
```

The returned answer is "P1 in [(0.0,0.0),(3.5,3.5)]" and "P2 in [(3.5,0.0),(20.0,16.5)]" (see Figure 7.2(a)). Note that real values are propagated from the real domain to the domain *real_point* by means of the constraints for `P1` and `P2`.

Figure 7.2: Information flow between different computation domains

[8] $[X_1, \ldots, X_n]$ in R is a shorthand for X_1 in $R, \ldots, X_n$ in R.

Of course, values can also be propagated in the reverse way (i.e., from the domain *real_point* to the real domain) by defining the adequate constraint operators. For example, we can extend the program shown above with the following declarations of operators.

```
    %----- Declarations of operators
:-dynamic ':&&:'/2.
:-op(625,fx,':&&:').
declara(:&&:,real_point,real).                                    (5)

:-dynamic ':**:'/2.
:-op(625,fx,':**:').
declara(:**:,real_point,real).
    %----- Operator definitions on the bracket domain
:&&:(B,B).      :**:(B.B).                                        (6)
    %----- Operator definitions on the real_point domain
:&&:((A,B),A):-lattice(real_point,(A,B)).                         (7)
:**:((A,B),B):-lattice(real_point,(A,B)).
```

This extension declares and defines the operators :&&: and :∗∗: as follows

$$\text{':\&\&:'} :: \mathit{real\_point}^s \to \Re^s \qquad \text{':**:'} :: \mathit{real\_point}^s \to \Re^s$$
$$:\&\&:(p) = \mathbf{a}\} \qquad :**:(p) = \mathbf{b}\}$$

for any $p = (\mathrm{a}, b)\mathbf{\}} \in \mathit{real\_point}^s$, and $\mathrm{a}, \mathrm{b} \in \Re$. Thus the operators :&&: and :∗∗: extract respectively the first and the second coordinate of a real point. As consequence values may be propagated from the real point domain to the real domain. For instance by means of the following interval constraints:

```
X1 in :&&:(min P1),:&&:(max P1),
Y1 in :**:(min P1),:**:(max P1),
X2 in :&&:(min P2),:&&:(max P2),
Y2 in :**:(min P2),:**:(max P2).
```

Of course it is possible to devise more complex cases. For instance by defining the operators :+: and :-: for the real point domain similarly as done for the domain *reint_point* in Section 7.5.1, the constraint $plus/3$ can be used in this domain. Then, if we add, to the query shown above, the constraint "plus(P1,P2,(5.0,5.0))", whose meaning is $P1 + P2 = (5.0, 5.0)$, the system returns the solution "P1 in [(0.0,0.0),(1.5,3.5)]" and P2 in [(3.5,1.5),(5.0,5.0)]" that is illustrated in Figure 7.2(b).

7.5.3 The Heart Diagnosis Problem

In this section we explain how the problem formulated in Section 5.5 on page 136 is solved in the clp($\mathcal{L}$) language. The category domain (called here *pvc*) and the state domain (called here *STATE*) are declared in our clp($\mathcal{L}$) program as follows:

```
%-------------- Pvc Domain -------------------------------------------
%% The domain 'pvc' (the domain of the postventricular contractions)
:-dynamic pvc/1.
pvcs(none).
pvcs(low).
pvcs(high).

lattice(Ele,pvc):- (pvcs(Ele);var(Ele)),!.
     %--- Cardinality
finite(pvc).
     %--- For finite domains we define the succesor and predecessor
succ(pvc,none,low).
succ(pvc,low,high).
succ(pvc,high,high).                %% succ(pvc,top,top).

pred(pvc,none,none).                %% pred(pvc,bottom,bottom).
pred(pvc,low,none).
pred(pvc,high,low).
     %--- Ordering
lt(normal,pvc,none,low).
lt(normal,pvc,none,high).
lt(normal,pvc,low,high).
     %--- Glb and Lub
glb(normal,pvc,X,Y,Z):- lattice(X,pvc),lattice(Y,pvc),!,
                         (lt(normal,pvc,X,Y) -> Z=X;Z=Y).
lub(normal,pvc,X,Y,Z):- lattice(X,pvc),lattice(Y,pvc), !,
                            (gt(normal,pvc,X,Y) -> Z=X;Z=Y).
     %--- Top and bottom elements
bottom(pvc,none).
top(pvc,high).

%-------------- State Domain -----------------------------------
:-dynamic state/1.
states(error1).
states(normal).
states(error2).
states(angina).
states(error3).
states(infarction).
states(error4).

lattice(Ele,state):- (states(Ele);var(Ele)),!.
     %--- Cardinality
finite(state).
```

```
     %--- For finite domains we define the succesor and predecessor
succ(state,error1,normal).
succ(state,normal,error2).
succ(state,error2,angina).
succ(state,angina,error3).
succ(state,error3,infarction).
succ(state,infarction,error4).
succ(state,error4,error4).      %% succ(state,top,top).

pred(state,error1,error1).
pred(state,X,Y):-succ(state,Y,X),!.
     %--- Ordering
lt(normal,state,A,A):-lattice(A,state),!,fail.
lt(normal,state,X,Y):-lattice(X,state),lattice(Y,state),
                      succ(state,X,Y),!.
lt(normal,state,X,Y):-lattice(X,state),lattice(Y,state),
                      succ(state,X,Z),
                      X\==Z,!,lt(normal,state,Z,Y).
lt(normal,state,X,Y):-lattice(X,state),lattice(Y,state),!,fail.
     %--- Glb and Lub
glb(normal,state,X,Y,Z):- lattice(X,state),lattice(Y,state),!,
                        (lt(normal,state,X,Y) -> Z=X;Z=Y).
lub(normal,state,X,Y,Z):- lattice(X,state),lattice(Y,state), !,
                          (gt(normal,state,X,Y) -> Z=X;Z=Y).
%--- Top and bottom elements
bottom(state,error1).
top(state,error4).
```

Now the heart domain is constructed from the *pvc* domain by simply adding the following two clauses to our $clp(\mathcal{L})$ program:

```
product_Direct(real,integer,reint_point).
product_Direct(pvc,reint_point,heart).
```

The declaration of the three operators used in the problem is shown below. In this $clp(\mathcal{L})$ code the operators $\diamond'$, $\diamond$ and $\bowtie$ are typed as `:<<>>:`, `:<>:` and `:><:` respectively[9].

```
     %------------------ Operator ':<<>>:' ---------------------
:-dynamic ':<<>>:'/2.
:-op(625,fx,':<<>>:').
     %--- Mode declaration
declara(:<<>>:,state,heart).       %% ':<<>>:' :: state^s -> heart^s
     %--- Def. on Bracket domain
```

[9]The declaration of $\bowtie$ has been smoothly simplified.

```
:<<>>:(open,open).
:<<>>:(close,close).
     %--- Definition
%%   For simplicity we take top_{Int}= 50000 and top_{reals}=1000000.0
%%                          bot_{Int}=0     and bot_{reals}=0.0
:<<>>:(error1,(low,(0.0,0))):-!. %%(low,(0.0,0))= bottom in heart^s
:<<>>:(normal,(none,(40.0,4800))):-!.
:<<>>:(error2,(none,(100.0,4800))):-!.
:<<>>:(angina,(none,(105.0,4800))):-!.
:<<>>:(error3,(none,(105.0,13500))):-!.
:<<>>:(infarction,(low,(105.0,13501))):-!.
:<<>>:(error4,(high,(1000000.0,50000))):-!.

     %------------------  Operator ':<>:'  ----------------------
:-dynamic ':<>:'/2.
:-op(625,fx,':<>:').
     %--- Mode declaration
declara(:<>:,state,heart).       %% ':<>:'  :: state^s -> heart^s
     %--- Def. on Bracket domain
:<>:(open,open).
:<>:(close,close).
     %--- Definition
:<>:(error1,(low,(0.0,0))):-!.
:<>:(normal,(none,(100.0,13500))):-!.
:<>:(error2,(none,(105.0,13500))):-!.
:<>:(angina,(none,(1000000.0,13500))):-!.
:<>:(error3,(low,(1000000.0,13500))):-!.
:<>:(infarction,(high,(1000000.0,50000))):-!.
:<>:(error4,(high,(1000000.0,50000))):-!.

     %------------------  Operator ':><:'  ----------------------
:-dynamic ':><:'/2.
:-op(625,fx,':><:').
     %--- Mode declaration
declara(:><:,heart,state).       %% ':><:' :: heart^s -> state^s
     %--- Def. on Bracket domain
:><:(open,open). :><:(close,close).
     %--- Definition
:><:(Reading,error1):-lattice(Reading,heart),
              (le(normal,heart,Reading,(high,(1000000.0,4799)));
               le(normal,heart,Reading,(high,(40.0,50000)))),!.
:><:(Reading,normal):-lattice(Reading,heart),
               ge(normal,heart,Reading,(none,(40.0,4800))),
               le(normal,heart,Reading,(none,(100.0,13500))),!.
```

```
:><:(Reading,erro2):-lattice(Reading,heart),
               ge(normal,heart,Reading,(none,(100.00000001,4800))),
               le(normal,heart,Reading,(none,(105.0,13500))),!.
:><:(Reading,angina):-lattice(Reading,heart),
               ge(normal,heart,Reading,(none,(105.0,4800))),
               le(normal,heart,Reading,(none,(1000000.0,13500))),!.
:><:(Reading,error3):-lattice(Reading,heart),
               ge(normal,heart,Reading,(none,(40.0,13501))),
               le(normal,heart,Reading,(high,(105.0,50000))),!.
:><:(Reading,infarction):-lattice(Reading,heart),
               ge(normal,heart,Reading,(low,(105.0,13501))),!.
:><:(Reading,error4):-lattice(Reading,heart), !.
```

The following step is to code the high level constraint $\approx^{\alpha}/2$. In Section 5.5.1, we show that this constraint is based on the operator $\gg$ that must also be declared in our clp($\mathcal{L}$) program. The declaration of this operator and the definition of the constraint $\approx^{\alpha}/2$ are shown below:

```
     %------------- declaration of operator  ----------------
:-dynamic ':>>:'/3.
:-op(625,xfx,':>>:').
     %--- Mode declaration
declara(:>>:,L,L1,L).       %% ':>>:' :: L^s \times L1^s -> L
     %--- Def. on Bracket domain
:>>:(open,_,open).
:>>:(close,B,B).
     %--- Def. on integer x real  (i.e.,L=integer,L1=real)
:>>:(Ele1,Ele2,Ele3):-lattice(Ele1,integer), lattice(Ele2,real),
                      Ele3 is  Ele1 + integer(round(Ele2)).
     %--- Def. on real x real  (i.e.,L=real,L1=real)
:>>:(Ele1,Ele2,Ele3):-lattice(Ele1,real), lattice(Ele2,real),
                      Ele3 is  Ele1 + Ele2.
     %--- Def. on heart X real^2.
:>>:((PVC,(PR,WBC)),(AlphaPR,AlphaWBC),(PVC,(PR1,WBC1))):-
                      lattice((PVC,(PR,WBC)),heart),
                      lattice((AlphaPR,AlphaWBC),real_point),
                      :>>:(PR,AlphaPR,PR1),
                      :>>:(WBC,AlphaWBC,WBC1).

    %----------- CONSTRAINT  'APPROXIMATELY EQUAL VIA ALPHA' ----
operator(650,xfx,~=:).
constraints (~=:)/2.
X ~=: (Y,Alpha) <=>
        (X in ((min Y) :>>: (min Alpha))..((max Y) :>>: (max Alpha)),
```

```
        Y in ((min X) :>>: (min Alpha))..((max X) :>>: (max Alpha))
       ).
```

The final step is to code the high level constraint *diagnostic*/3 whose *clp*($\mathcal{L}$) code is shown below.

```
constraints diagnostic/3.
diagnostic(Reading,State,Alpha) <=>
          ~=:(Reading,(TrueRead,Alpha)),
          State in (:><:(min TrueRead))..(:><:(max TrueRead)),
          TrueRead in (:<<>>:(min State))..(:<>:(max State)).
```

The problem is solved by calling this last constraint. In the following the query marked in Section 5.5.2 (i.e., the constraint set $\{c_1, c_2, c_3\}$) can be solved by typing the equivalent constraints in our system. Below, we show[10] how this is solved[11].

```
?- Alpha in ((-1.0,-45.7),close)..((1.0,45.7),close),
   Reading in ((none,(85.6,10000)),close)..((none,(85.6,10000)),close),
   diagnostic(Reading,State,Alpha).

            %%ANSWER
   State = normal,
   Reading = (none,85.6,10000),
   Alpha in((-1.0,-45.7),close)..((1.0,45.7),close),
   _A in ((none,84.6,9954),close)..((none,86.6,10046),close) ? ;
no.
```

7.6 Related and Further Work

The *clp*($\mathcal{L}$) language is based on the indexical model and, in recent years, the indexical approach has been successfully used to implement a number of CLP languages. Well-known CLP systems such as SICStus (Sicstus manual, 1994) or IF/Prolog (If/Prolog, 1994) now integrate the primitive *x in r* to solve constraints on the finite domain (FD). Traditionally, indexicals have been implemented by modifying the WAM (Aït-kaci, 1999) but we have to consider alternative approaches that may lead to more efficient implementations. In this section we briefly discuss some of the traditional existing implementations of the indexical model as well as alternative approaches for their implementation.

[10] Observe that the query is called from the SICStus command line since the syntax analyser of our interpreter is not fully implemented.

[11] The variable _A corresponds to the variable TrueRead.

7.6.1 WAM Based Implementations

Originally, (Diaz and Codognet, 1993) integrated, by extending the WAM, the primitive x *in* r in the logical language clp(FD) to solve interval constraints on the finite domain (an extended and more recent version of this paper is found in (Codognet and Diaz, 1996a)). Also, the indexical approach was applied in the Boolean domain by encoding a Boolean solver at a low level with the basic mechanism of clp(FD) and where Boolean constraints (e.g., and, or, not, etc.) were decomposed in primitive x *in* r constraints. This extension of the clp(FD) system was called clp(B/FD) (Codognet and Diaz, 1993). The success of this system was surprising since sometimes showed a better performance than most of the existing specific Boolean solvers.

The indexical approach of the clp(B/FD) system was improved by specialising this system for just the Boolean domain (Codognet and Diaz, 1994). The specialisation introduced a new type for the Boolean domain and new instructions for the Boolean variables. As result, the clp(B) system was born. This system is a Boolean constraint solver based on local propagation techniques and on the indexical approach that also follows the glass box approach of compiling high level constraints into primitives constraints (as done in the clp(FD) system). (Codognet and Diaz, 1994) and (Codognet and Diaz, 1996b) shown that the clp(B) solver is almost one magnitude of order faster than most of the existing Boolean constraint solvers. Surprisingly, as it was shown in (Codognet and Diaz, 1994), "the low level primitive constraint which is at the core of clp(B) can be implemented into a WAM-based logical engine with a minimal extension: only four new instructions are needed". This means that this Boolean solver can be "easily" integrated into any Prolog system.

As another development, (Georget and Codognet, 1998) used the indexical approach to implement the language clp(FD,S). This language is a generic schema for compiling semiring-based constraints and corresponds to the implementation of the generic framework for FD constraint satisfaction and optimisation defined in (Bistarelli et al., 1995; Bistarelli et al., 1997b) (see Section 4.7.2 on page 120 for more information). Because of the generality of the approach, some optimisations that could be introduced in the solver were lost. Despite this, the clp(FD,S) was still found to be fairly efficient with respect to dedicated systems.

More recently, in (Goualard et al., 1999) it is demonstrated that is possible to implement an efficient solver (called *DecLic*) over continuous domains (i.e. the real domain) by extending the clp(FD) language.

7.6.2 An Alternative Approach: the ATOAM Model

It should be noted that other models have be considered for future implementation of the indexicals, as for example the model of B-Prolog. As already declared in Section 2.5, B-Prolog is based on a new abstract machine called ATOAM (Zhou et al., 1990; Zhou, 1994) (yet Another matching Tree Oriented Abstract Machine) that is an alternative approach to the classical WAM used in most of the existing logic languages.

In the ATOAM arguments are passed through an stack, on reverse to the WAM

approach of clp(FD) (Codognet and Diaz, 1996a) where arguments are passed in registers. The access to register is faster than the access to memory and thus the WAM approach seems to be more efficient. However, the WAM scheme requires the argument registers to be saved and restored for backtracking and makes it difficult to implement full tail recursion elimination. The solution is to have emulator-based implementations because registers are actually simulated by using memory. As already noted, parameter passing and control stack management are two crucial issues in the efficiency of Prolog implementations. The ATOAM is a new abstract machine that contributes to the Prolog-implementation development with new characteristics. There are three main innovations with respect to the WAM:

1. Arguments are passed directly into stack frames.
2. Only one frame is used for each procedure call (whereas the WAM used two frames).
3. Procedures are translated into matching trees if possible, and clauses in each procedure are indexed on all input arguments.

The ATOAM solves the problems of WAM since tail recursion can be handled in most cases like a loop statement in procedural languages because backtracking requires less bookkeeping operations.

The B-Prolog system (at least till version 4.0) supports the delaying (co-routining) mechanism, which can be used to implement concurrency, test-and-generate search algorithms, and most importantly constraint propagation algorithms. This is a non-standard delaying mechanism that relaxes the strict left-to-right computation rule adopted in Prolog and enables the execution of some predicates calls to be delayed until some variables in them are instantiated. For specifying delay, B-Prolog includes a special kind of construct called *delay clauses* that have the form

delay Head :- Condition : [{Triggers}] Action.

where Condition is a sequence of in-line tests, Triggers is a (possibly empty) sequence of trigger declarations, and Action is a sequence of arbitrary calls. For any call, if this matches the Head and Condition is satisfied, then the call delays and Action is executed. The call will be kept unchanged before Action is executed. If the call does not match Head or Condition fails, then other clause is tried. If Action fails, then the original predicate call will fail. The delay clauses are different from the guard clauses in concurrent languages, and the ':' does not mean commitment. After a delay clause is chosen, the predicate can be reentered and then remaining clauses can be retried later (the re-execution of the clauses is declared by the sequence of trigger declarations). See (Zhou, 2000b, Chapter 10) for more information.

For instance, indexicals can be implemented by using delay clauses. Consider the following indexical[12]:

[12]This example is taken from (Zhou, 2000b, Page 36-37)

```
            X in min(Y)+min(Z)..max(Y)+max(Z).
```

The following clauses implement this indexical in B-Prolog:

```
delay 'V in V+V'(X,Y,Z):-dvar(X) :
                {ins(Y),min(Y),max(Y),ins(Z),min(Z),maz(Z)},
                consistency_check_v_vv(X,Y,Z).
'V in V+V'(X,Y,Z):-true :
                consistency_check_v_vv(X,Y,Z).

consistency_check_v_vv(X,Y,Z) :-
                fd_min_max(Y,MinY,MaxY),
                fd_min_max(Z,MinZ,MaxZ),
                MinX is MinY+MinZ,
                MaxX is MaxY+MaxZ,
                X in MinX..MaxX.
```

The constraint is propagated whenever a bound in Z or Y is changed. However, note that this constraint propagates from Y and/or Z to X but not on reverse.

The delay clauses of B-Prolog are powerful constructs that also enable the implementation of specific constraints in the user level such as reified constraints. As a consequence, the approach of B-Prolog should be considered as an alternative for future implementations of our system.

7.7 Concluding Remarks

In this chapter we have introduced the $clp(\mathcal{L})$ language and shown the main characteristics of a first prototype implementation.

The $clp(\mathcal{L})$ language gives support for interval constraint solving on any set of (possibly user-defined) lattices and thus it provides a uniform approach for interval constraint solving on the usual domains such as reals, finite ranges of integers, sets and Booleans among others, and on more application-specific domains defined by the user. Also, as the theoretical framework is based on lattice theory, it is straightforward to construct new domains and new constraints for these domains from existing ones. Moreover, as the operators can be defined over more than one domain, information can flow between different computation domains.

In this chapter we have shown how the user can define new domains from the scratch or from the combination of existing ones, constraint operators through information can flow between the computation domains and (generic or overloaded) high level constraints for specific applications. We have also described briefly the existing prototype implementation of the $clp(\mathcal{L})$ language and shown, by means of examples, the flexibility, transparency and genericity of the language in the formulation of problems as well as the cooperative character of the $clp(\mathcal{L})$ system.

We have not discussed performance of our prototype since the implementation described here is built with CHRs and although highly expressive are known to be inefficient (see Chapter 3). However, the indexical approach has been proved to be very efficient over both the finite and continuous domains (Codognet and Diaz, 1996a; Goualard et al., 1999) so we anticipate that we can adapt the techniques used for the implementation of systems such as clp(FD), clp(B) and *DecLic* to our constraint system thereby obtaining performance comparable with other CLP systems. Despite the expected loss of some optimizations for specific domains due to the generality of our framework, we expect to obtain competitive performance compared to domain-specific systems.

Currently, the branching schema described in Chapter 6 is not integrated in the prototype implementation. This is an issue of further work.

7.8 Contributions

The main contributions of this chapter are listed below.

- Firstly, it demonstrates that the theoretical ideas shown in Part III are feasible.
- Secondly, it demonstrates that a single system may provide support for user and system defined constraints, user and system defined domains (independent of their cardinalities is infinite), interaction between these domains and interval constraint solving defined on the set of computation domains.
- Finally it proposes a novel interval CLP language combining a number of interesting characteristics such as generic and overloaded constraints, solver cooperation, transparency to define both constraints and domains and complete and/or partial interval constraint solving.

  To our knowledge no existing interval CLP language groups together all these characteristics.

Chapter 8

Concluding Remarks

Roma locuta est; causa finita est
Roma has spoken; the case is concluded

Traditional summary of words found in
Sermons (Antwerp, 1702), n.131, seat 10
St. Augustine of Hipo, Ad. 354-430

8.1 Summary of the Results

In this thesis, we have proposed an interval constraint solving schema that combines three desirable characteristics: (1) true generality, (2) cooperativity of solvers and (3) full transparency in the definition of constraints, domains and the propagation mechanism. The schema has been developed in several stages. Also, as the schema is based on interval reasoning, we can forecast a fourth desirable characteristic: efficiency in constraint solving.

We first compared the different glass box approaches over the FD. The reason the FD was elected for this comparison is that most of the existing glass box systems have been designed for this domain. Moreover, glass box systems also allow user extensions of the built-in system. From the comparison, we have observed that glass box systems provides higher flexibility than black box systems without any significant loss of efficiency. As a result we have used the indexical model as the base on which to construct the foundations of our schema. This choice has been justified in that the indexical model has an acceptable performance while providing a high flexibility in the formulation of (discrete) problems and some transparency in the definition of new constraints.

We generalised the indexical model to any domain with lattice structure to provide a generic schema for interval constraint propagation that is valid for any lattice, for any cardinality. We have also shown that the schema is useful for most of the classical

domains as well as for new domains resulting either from the application of lattice combinators over existing domains or from user definitions. We have proved the main properties of the schema such as correctness. Termination of the propagation can be assured by the means of a precision map required for each computation domain. Monotonicity of constraints is also assured by the intrinsic formalisation of the schema. As consequence non-monotonic constraints are easily detected *a priori* (i.e., before their resolution) by simply adding a test about the validity of the constraints in the theoretical framework. Moreover, as in the indexical model for the FD, our propagation framework is based on a single primitive constraint $x \sqsubseteq r$ which provides both a specification of the constraint and some control over its propagation mechanism.

Since the framework for constraint propagation provides support for any set of lattice-structure computation domains, the primitive constraint makes use of constraint operators defined on multiple domains. This makes possible a novel class of solver cooperation allowing information to flow between the computation domains supported by the framework (i.e., any lattice). The concept of high level constraints for CLP(FD) has also been generalised to lattices. This generalisation increases the potential for solver cooperation and also allows the definition of generic and overloaded constraints. Observe also that the cooperation of solvers in our framework guarantees the property of *type security*. In fact, constraint operators behave as the "interface" of communication (of the solvers) from and to which information is propagated. As already indicated above, non-valid constraints can be detected *a priori* and, as consequence, only cooperating constraints (i.e., those involving constraints operators defined on some argument different from the computation domain over which they are defined) that are valid in the framework are allowed for their resolution.

The generic and cooperative propagation schema for interval constraint solving has been extended. The resulting schema is also transparent, cooperative and generic and thus can be used to solve classical CSPs as well as partial CSPs. The resulting schema, called the branching schema is valid for any set of lattices and is a generalisation of the well known method split-and-solve of CLP(BNR) that solves interval constraints in the real, integer and Boolean domain. The branching schema is parametrised in a number of procedures that have been partially specified (i.e., we have declared them formally and defined both their pre- and post-conditions). Then, we have also studied a number of interesting properties that subject to certain conditions held by the main procedures of the schema are satisfied by any instances of the schema. Termination is again guaranteed by means of the precision map concept used in the propagation schema.

To demonstrate the feasibility of our theoretical framework we have proposed a CLP language based on the framework using the syntax and structure of Prolog. We have described the main characteristics of both the language and the prototype. We have used our language to solve several example problems. These illustrate the flexibility of the language for the formulation of CSPs. They show that systems constructed from our theory are easy to learn and manage.

8.2 Summary of Main Contributions

This thesis has dealt with a number of different aspects of CLP. In each chapter, we have summarised its main contributions. We now present a brief summary of the most important contributions of the thesis.

- We have compared the efficiency and some aspects of the expressiveness of eight existing constraint systems on the FD. To our knowledge, this is the first time that this number of systems has been compared in the constraint setting. This comparison discusses some of the issues that need to be taken into account when choosing a system for solving a specific application in the discrete domain. It also provides a starting point for further comparative work.

- We have proposed a generalisation of the indexical model for the FD to domains with lattice structure, independent of their cardinality. This generic model provides a framework for interval constraint solving. We have proved a number of interesting properties that are satisfied by any (possibly cooperative) instance of this framework. The proposal is a glass box system for both domains and constraints with the possibility to connect solvers defined on different domains. As an immediate result, due to its flexibility and genericity our framework provides an alternative to the CHR language. Moreover, as it is based on the indexical model which has been proved to be very efficient compared to the CHR model, we expect that future implementations based on other more direct implementations of indexicals to have improved performance.

- We have also demonstrated that solver cooperation can be integrated transparently in our system what means that a single system may provide support for a glass box solver cooperation mechanism. Perhaps the main drawback of this integration may be the loss of efficiency but we think that this can be compensated by a gain in expressiveness and flexibility.

8.3 Further Work

Although we have achieved the objectives of Chapter 1, there are a number of open lines to continue with (or perhaps we should say "complement") the work carried out in this thesis. We outline some of them here.

With respect to our comparison of FD constraint systems, it should be noted that we have only compared the FD libraries. Similar comparative studies could also be made for other common domains such as intervals (Benhamou, 1995) or reals (Jaffar et al., 1992b). Alternative approaches such as integer linear programming could also be considered for a comparison between systems. This could be the subject of future work.

In our theoretical framework for interval constraint solving we have imposed that the sets of constrained variables associated with each computation must be disjoint. However, it should be possible to remove such a restriction and consider sub-lattices of

lattices as computation domains. These could provide a means of having variable sets of a sublattice allowed as variables in the main lattice. This is another topic for future work.

We have studied generic properties of interval branching in interval constraint solving. To guarantee these properties we have imposed a number of generic conditions over the heuristics for value and variable ordering. However, these conditions do not guarantee the minimality of the search space, that is to say, they guarantee termination, correction and completeness but do not avoid the redundancy of information in the search tree. Therefore, it would be interesting to study if there exist any properties of the heuristics used in the interval branching process that lead to a reduced search space. For instance, for the splitting functions (see Definition 6.11) we could recommend the following property:

$$\forall i, j \in \{1, \ldots, k\}, i \neq j : glb_{\mathcal{C}_L^X}(c_i, c_j) \text{ is inconsistent}$$

that should ensure that "partitions share no common element" and hence avoiding the redundancy of information in different branches of the search tree. Of course this requires further study.

Observe also, that in the schema $branch_{\alpha+}/3$ (i.e., the extension of the basic branching schema to solve optimization problems) we had not taken into account the efficiency and just considered the given cost function and the ordering relation in its co-domain for finding an optimal solution. Obviously, this is clearly inefficient since the whole search tree has to be traversed completely. Of course, our aim was to study a set of generic properties (such as correctness, completeness and termination) held by our branching schema during the solving of optimisation problems. In spite of the fact that we had not in mind the efficiency issue, it would be interesting to investigate if this schema can be reformulated (if possible) to prune the search tree when it is known that no better solution, than the solution found so far, is in the current subtree.

The implementation described here is built with CHRs and although highly expressive are known to be inefficient. Thus, an issue of further work is to integrate our system into another more efficient architecture to better evaluate its efficiency. In this sense we are considering the construction of an abstract machine for the framework proposed here and, of course, its further implementation in a CLP language. Initially, we think that this implementation would consist of extending the WAM as done in the implementation of the clp(FD) system (Diaz and Codognet, 1993). However, an alternative approach is to use B-Prolog for the implementation. This is based on new abstract machine called ATOAM that has been shown to be fairly efficient. Therefore, initially we need to further investigate which is the best approach for the abstract machine of our system and then, to implement it.

We plan to compare our system with constraint systems dedicated to specific computation domains. Once we have a improved implementation, we plan to "measure" its performance and expressiveness with respect to existing systems. We anticipate that as the framework is based on the indexical model (and this has been shown to be very efficient) it is expected that a good implementation will be comparable with that of

dedicated solvers on specific domains. Note also that this implementation should show a significantly better performance that the best implementations of the CHR language on top of well known CLP systems.

We are currently studying the integration of our interval constraint solving schema into a functional logic language. These languages combine logic properties with functional characteristics offering an adequate setting for our generic solver. We believe that there exists a strong analogy between types and functions in functional logic languages and computation domains and constraint operators, respectively, in our framework. A direct consequence of these analogies is that our generic schema could be smoothly integrated in these languages. Some preliminary work has already been done in functional logic language Toy (Caballero et al., 1997), although the work is far from maturity (Cazorla, 2001).

Bibliography

Abdennadher, S. (1997). Operational semantics and confluence of constraint propagation rules. In Smolka, G., editor, *3rd International Conference on Principles and Practice of Constraint Programming (CP'97)*, number 1330 in LNCS, pages 252–266, Linz, Austria. Springer-Verlag.

Abdennadher, S., Frühwirth, T., and Meuss, H. (1999). Confluence and semantics of constraint simplification rules. *Constraints*, 4(2):133–165.

Aggoun, A. and Beldiceanu, N. (1992). Extendig CHIP to solve complex scheduling and placement problems. In *Journées Francophones de Programmation Logique (JFPL'92)*, Lille, France.

Aggoun, A., Chan, D., Dufresne, P., Falvey, E., Grant, H., Herold, A., Macartney, G., Meier, M., Miller, D., Mudambi, S., Perez, B., Rossum, E. V., Schimpf, J., Periklis, Tsahageas, A., and de Villeneuve, D. (1995). ECL^iPS^e 3.5, user manual. European Computer -Industry Research Centre (ECRC). Munich.

Aiba, A. and Sakai, K. (1989). CAL: a theoretical background of constraint logic programming and its applications. *Journal of Symbolic Computation*, 8:589–603.

Aiba, A., Sakai, K., Sato, Y., Hawley, D., and Hasegawa, R. (1988). The constraint logic programming language CAL. In ICOT, editor, *International Conference on Fith Generation Computer Systems (FGCS'88)*, pages 263–276, Tokyo, Japan. Ohmsha Ltd. and Springer-Verlag.

Aït-kaci, H. (1999). *Warren's Abstract Machine: a tutorial reconstruction*. The MIT Press, Cambridge, MA.

Aït-kaci, H., Lincoln, P., and Nasr, R. (1987). Le Fun: logic, equations and functions. In *1987 Symposium on Logic Programming (SLP'87)*, pages 17–23, San Francisco, California. IEEE-CS.

Aït-kaci, H. and Nasr, R. (1986). LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3(3):185–215.

Aït-kaci, H. and Podelski, A. (1993). Towards a meaning of LIFE. *Journal of Logic Programming*, 16(3):195–234. A preliminary version appeared in (Maluszynski and Wirsing, 1991), pp:255-274.

Aït-kaci, H., Podelski, A., and Smolka, G. (1994). A feature constraint system for logic programming with entailment. *Theoretical Computer Science*, 122(1-2):263–283.

Alefeld, G. and Herzberger, J. (1983). *Introduction to interval computations*. Academic Press, London and San Diego.

Allen, J. (1983). Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843.

Apt, K. (1999). The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210.

Arenas, P., Gil, A., and López-Fraguas, F. (1994). Combining lazy narrowing with disequality constraints. In *6th International Symposium on Programming Languages Implementation and Logic Programming (PLILP'94)*, number 844 in LNCS, pages 385–399, Madrid, Spain. Springer-Verlag.

Arenas, P., Hortalá, M., López-Fraguas, F., and Ullán, E. (1996). Real constraints within a functional logic language. In Lucio, P., Martelli, M., and Navarro, M., editors, *Joint Conference on Declarative Programming (APPIA-GULP-PRODE'96)*, Donostia-San Sebastian, Spain.

Arenas, P., López-Fraguas, F., and Rodríguez-Artalejo, M. (1999). Functional plus logic programming with built-in and symbolic constraints. In Nadathur, G., editor, *International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, number 1702 in LNCS, pages 152–169, Paris, France. Springer-Verlag.

Arnon, D., Collins, G., and McCallum, S. (1984). Cylindrical algebraic decomposition I: the basic algorithm. *SIAM Journal on Computing*, 13(4):865–877.

Azevedo, F. and Barahona, P. (2000). Modelling digital circuits problems with set constraints. In Lloyd, J. W., Dahl, V., Furbach, U., Kerber, M., Lau, K., Palamidessi, C., Pereira, L., Sagiv, Y., and Stuckey, P. J., editors, *1st International Conference on Computational Logic (CL2000)*, number 1861 in LNCS, pages 414–428, London, UK. Springer-Verlag.

Baader, F. and Schulz, K. (1995). On the combination of symbolic constraints, solution domains and constraints solvers. In Montanari, U. and Rossi, F., editors, *1st International Conference on Principles and Practice of Constraint Programming (CP'95)*, number 976 in LNCS, pages 380–397, Cassis, France. Springer-Verlag.

Baader, F. and Schulz, K. (1996). Unification in the union of disjoint equational theories: combining decision procedures. *Journal of Symbolic Computation*, 21(2):211–243.

Baader, F. and Schulz, K. (1998). Combination of constraint solvers for free and quasi-free structures. *TCS*, 192(1):107–161.

Badía, J. (2000). Evaluación teórica y práctica de algoritmos de arco consistencia en problemas de resolución de restricciones. Master's thesis, ETSI (Informática) Málaga University, Málaga, Spain. Directed by Antonio J. Fernández (in spanish).

Bagnara, R., Gori, R., Hill, P., and Zaffanella, E. (2001). Finite-tree analysis for constraint logic-based languages. In Cousot, P., editor, *8th International Symposium on Static Analysis (SAS'01)*, volume 2126 of *LNCS*, pages 165–184, Paris, France. Springer-Verlag, Berlin.

Barendregt, H. (1984). *The lambda calculus - Its syntax and semantics*. Studies in Logic and the Foundations of Mathematics, 103. North-Holland, Netherlands. Second Revised Edition.

Barendregt, H. (1990). Functional programming and lambda calculus. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 323–363, Nederlands. Elsevier. J. van Leeuwen editor.

Barták, R. (1999). Constraint programming: In pursuit of the holy grail. In *8th Annual Conference of Doctoral Students WDS'99 (invited lecturer)*, Troja, Prague. Charles University.

Barth, P. (1994). Short guide to CLP($\mathcal{PB}$). Max-Planck-Institut für Informatik. Available at *ftp://www.mpi-sb.mpg.de/pub/tools/CLPPB/clppb.html*.

Barth, P. (1996). *Logic-based 0-1 constraint Programming*. Operations Research/Computer Science Interfaces. Kluwer.

Barth, P. and Bockmayr, A. (1996). Modelling 0-1 problems in CLP($\mathcal{PB}$). In Wallace, M., editor, *2nd International Conference on the Practical Application of Constraint Technology (PACT'96)*, pages 1–9, London, UK. Prolog Management Group.

Benhamou, F. (1993). Boolean algorithms in Prolog III. In *(Benhamou and Colmerauer, 1993)*, pages 307–325, Cambridge, Massachusetts, London, England. The MIT Press.

Benhamou, F. (1995). Interval constraint logic programming. In Podelski, A., editor, *Constraint Programming: Basics and Trends*, number 910 in LNCS, pages 1–21, Châtillon-sur-Seine, France. Springer-Verlag.

Benhamou, F. (1996). Heterogeneous constraint solving. In Hanus, M. and Rodríguez-Artalejo, M., editors, *5th International Conference on Algebraic and Logic Programming (ALP'96)*, number 1139 in LNCS, pages 62–76, Aachen, Germany. Springer-Verlag.

Benhamou, F. and Colmerauer, A., editors (1993). *Constraint logic programming: selected research*. The MIT Press, Cambridge, MA.

Benhamou, F., Goualard, F., and Granvilliers, L. (1997). Programming with the declic language. In *2nd International Workshop on Interval Constraints*, Port-Jefferson, NY, USA.

Benhamou, F., Goualard, F., Granvilliers, L., and Puget, J.-F. (1999). Revising hull and box consistency. In D. De Schreye, editor, *16th International Conference on Logic Programming (ICLP'99)*, pages 230–244, Las Cruces, New Mexico, USA. The MIT Press.

Benhamou, F., McAllester, D., and Van Hentenryck, P. (1994). CLP(Intervals) revisited. In Bruynooghe, M., editor, *4th International Symposium on Logic Programming (ILPS'94)*, Logic Programming, pages 124–138, Ithaca, New York. The MIT Press.

Benhamou, F. and Older, W. (1997). Applying interval arithmetic to real, integer and Boolean constraints. *The Journal of Logic Programming*, 32(1):1–24.

Biasizzo, A. and Novak, F. (1995). Model-based diagnosis of analog circuits. In *International Mixed Signal Testing Workshop*, pages 95–100, Grenoble, France.

Bistarelli, S., Montanari, U., and Rossi, F. (1995). Constraint solving over semirings. In *14th International Joint Conference on Artificial Intelligent (IJCAI'95)*, pages 624–630, Québec, Canada. Morgan Kaufman.

Bistarelli, S., Montanari, U., and Rossi, F. (1997a). Semiring-based constraint logic programming. In *15th International Joint Conference on Artificial Intelligent (IJCAI'97)*, Nagoya, Japan. Morgan Kaufman.

Bistarelli, S., Montanari, U., and Rossi, F. (1997b). Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236.

Bistarelli, S., Montanari, U., Rossi, F., Schiex, T., Verfaillie, G., and Fargier, H. (1999). Semiring-based CSPs and valued CSPs: frameworks, properties and comparison. *Constraints*, 4(3):199–240.

Bockmayr, A. (1993). Logic programming with pseudo-Boolean constraints. In *(Benhamou and Colmerauer, 1993)*, pages 327–350, Cambridge, MA. The MIT Press.

Bockmayr, A. (1994). Solving pseudo-Boolean constraints. In Podelski, A., editor, *Constraint Programming: Basics and Trends*, number 910 in LNCS, pages 22–38, Châtillon-sur-Seine, France. Springer-Verlag.

Buchberger, B. (1987a). Applications of Gröbner bases in non-linear computational geometry. In *Trends in Computer Algebra*, number 296 in LNCS. Springer-Verlag.

Buchberger, B. (1987b). Solving problems in non-linear computational geometry by Gröbner bases. In *IMA Workshop on Mathematical Aspects of Scientific Software (invited paper)*, New York. Springer.

Buchberger, B. (1997). Gröbner bases: an introduction. In Kuich, W., editor, *19th International Colloquium on Automata, Languages and Programming (ICALP'92)*, number 623 in LNCS, pages 378–379, Vienna, Austria. Springer-Verlag.

Büttner, W. and Simonis, H. (1997). Embedding Boolean expressions into logic programming. *Journal of Symbolic Computation*, 4(2):191–205.

Caballero, R., López-Fraguas, F., and Sánchez, J. (1997). User's manual for $\mathcal{TOY}$. Technical report SIP-5797, Universidad Complutense de Madrid, Dpto. Lenguajes, Sistemas Informáticos y Programación.

Carlson, B., Carlsson, M., and Diaz, D. (1994a). Entailment of finite domain constraints. In Van Hentenryck, P., editor, *11th International Conference on Logic Programming (ICLP'94)*, pages 339–353, Santa Margherita Ligure, Italy. The MIT Press.

Carlson, B., Janson, S., and Haridi, S. (1994b). AKL(FD): A concurrent language for FD programming. In Bruynooghe, M., editor, *4th International Symposium on Logic Programming (ILPS'94)*, Logic Programming, pages 521–535, Ithaca, New York. The MIT Press.

Carlsson, M. and Brindal, M. (1993). Automatic frequency assignment for cellular telephones using constraint satisfaction techniques. In Warren, D., editor, *10th International Conference on Logic Programming (ICLP'93)*, pages 647–665, Budapest, Hungary. The MIT Press.

Carlsson, M., Ottosson, G., and Carlson, B. (1997). An open-ended finite domain constraint solver. In Montanari, U. and Rossi, F., editors, *9th International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'97)*, number 1292 in LNCS, pages 191–206, Southampton, UK. Springer-Verlag.

Carro, M., Hermenegildo, M., Bueno, F., Cabeza, D., García, M., López, P., and Puebla, G. (2000). An introductory course of constraint logic programming. Computer Science School, Technical University of Madrid, UPM.

Cazorla, J. (2001). Programación lógico-funcional con restricciones genéricas. Master's thesis, ETSI (Informática) Málaga University, Málaga, Spain. Directed by Antonio J. Fernández (in spanish).

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

CLP(BNR) (1988). *CLP(BNR) reference and users manuals.* Bell Northern Research, Ottawa, Ontario, Canada.

Codognet, P. and Diaz, D. (1993). Boolean constraint solving using clp(FD). In Miller, D., editor, *1993 International Symposium on Logic Programming (ILPS'93)*, pages 525–539, Vancouver, British Columbia, Canada. The MIT Press.

Codognet, P. and Diaz, D. (1994). clp(B): combining simplicity and efficiency in Boolean constraint solving. In *6th International Symposium on Programming Languages Implementation and Logic Programming (PLILP'94)*, number 844 in LNCS, pages 244–260, Madrid, Spain. Springer-Verlag.

Codognet, P. and Diaz, D. (1996a). Compiling constraints in clp(FD). *The Journal of Logic Programming*, 27(3):185–226.

Codognet, P. and Diaz, D. (1996b). A simple and efficient boolean solver for constraint logic programming. *The Journal of Automated Reasoning*, 17(1):97–129.

Cohen, J. (1988). A view of the origins and development of Prolog. *Communications of the ACM*, 31(1):26–36.

Cohen, J. (1990). Constraint logic programming languages. *Communications of the ACM*, 33(7):52–68.

Colmerauer, A. (1990). An introduction to PROLOG III. *Communications of the ACM (CACM)*, 33(7):69–90.

Colmerauer, A. and Roussel, P. (1993). The birth of Prolog. *ACM SIGPLAN Notices as part of 2nd ACM SIGPLAN Conference on History of Programming Languages*, 28(3):37–52. Cambridge, United States.

Cras, J.-Y. (1993). A review of industrial constraint solving tools. *AI Intelligence*.

Crossley, J., Mandel, L., and Wirsing, M. (1996). First-order constrained lambda calculus. In Baader, F. and Schulz, K. U., editors, *1st International Workshop on Frontiers of Combining Systems (Frocos'96)*, volume 3 of *Applied Logic Series*, pages 339–356, Munich, Germany. Kluwer Academic Publishers.

Csontó, J. and Paralič, J. (1997). A look at CLP: theory and application. *Applied Artificial Intelligence*, 11:59–69.

Curtis, S., Smith, B., and Wren, A. (2000). Constructing driver schedules using iterative repair. In *2nd International Conference on The Practical Applications of Constraint Technology and Logic Programming (PACLP'2000)*, pages 59–78, Manchester, UK. Practical Application Company.

Davey, B. and Priestley, H. (1990). *Introduction to lattices and order*. Cambridge University Press, Cambridge, England.

Davis, E. (1987). Constraint propagation with interval labels. *Artificial Intelligence*, 32(3):281–331.

de Boer, F. and Palamidessi, C. (1994). From concurrent logic programming to concurrent constraint programming. In *(Levi, 1994)*, pages 55–113. Oxford University Press.

de la Banda, M., Jeffery, D., Marriott, K., Nethercote, N., Stuckey, P., and Holzbaur, C. (2001). Building constraint solvers in HAL. In *International Conference on Logic Programming (ICLP'2001)*, Paphos, Cyprus. To appear.

Demoen, B., de la Banda, M., Harvey, W., Marriott, K., and Stuckey, P. (1999a). Herbrand constraint solving in HAL. Technical Report 1999/49, Monash University, Melbourne.

Demoen, B., de la Banda, M., Harvey, W., Marriott, K., and Stuckey, P. (1999b). Herbrand constraint solving in HAL. In D. De Schreye, editor, *16th International Conference on Logic Programming (ICLP'99)*, pages 260–274, Las Cruces, New Mexico, USA. The MIT Press.

Diaz, D. (1994). clp(FD) 2.21 user's manual. INRIA-Rocquencourt, France.

Diaz, D. (1995). *Etude de la compilation des langages logiques de programmation par contraintes sur les domaines finis: le système clp(FD).* PhD thesis, l'université d'Orléans.

Diaz, D. and Codognet, P. (1993). A minimal extension of the WAM for clp(FD). In Warren, D., editor, *10th International Conference on Logic Programming (ICLP'93)*, pages 774–790, Budapest, Hungary. The MIT Press.

Diaz, D. and Codognet, P. (2000). GNU Prolog: beyond compiling Prolog to C. In Pontelli, E. and Costa, V., editors, *2nd International Workshop on Practical Aspects of Declarative Languages (PADL'2000)*, number 1753 in LNCS, pages 81–92, Boston, USA. Springer-Verlag.

Dincbas, M., Simonis, H., and Van Hentenryck, P. (1988a). Solving a cutting-stock problem in constraint logic programming. In Kowalski, R. and Bowen, K., editors, *5th International Conference and Symposium of Logic Programming (ICLP/SLP'88)*, pages 42–58, Seattle, Washington. The MIT Press.

Dincbas, M., Simonis, H., and Van Hentenryck, P. (1990). Solving large combinatorial problems in logic programming. *Journal of Logic Programming*, 8(1):75–93.

Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T., and Berthier, F. (1988b). The constraint logic programming language CHIP. In ICOT, editor, *International Conference on Fith Generation Computer Systems (FGCS'88)*, pages 693–702, Tokyo, Japan. Ohmsha Ltd. and Springer-Verlag.

Dovier, A., Omodeo, E., Pontelli, E., and Rossi, G. (1996). {log}: a language for programming in logic with finite sets. *Journal of Logic Programming*, 28(1):1–44.

Erlt, M. and Krall, A. (1992). High level constraints over finite domains. Tech Report TR-1851-1992-14, Institute für Computer Sprachen, Technische Universität Wien, Austria.

Faugere, J.-C. (1994). *Résolution des systèmes d'équations algébriques.* PhD thesis, Université Paris 6.

Fernández, A. (1997). Towards a glass-box typed CLP language. In Fisher, M., editor, *Workshop on Automated Reasoning (WAR'97)*, AISB Workshop and Tutorial Series, pages 7–8, Manchester. University of Manchester.

Fernández, A. (1998). Srq solutions. *http://www.lcc.uma.es/∼afdez/srq.*

Fernández, A. (2000). clp(L) version 0.21, user manual. Available at *http://www.lcc.uma.es/∼afdez/generic.*

Fernández, A. and Hill, P. (1997a). Boolean and finite domain solvers compared using self referential quizzes. In Falaschi, M., Navarro, M., and Policriti, A., editors, *Joint Conference on Declarative Programming (APPIA-GULP-PRODE'97)*, pages 533–544, Grado, Italy. CLEUP. Also available as Technical Report ref.97.03, School of Computer Studies, University of Leeds, January, 1997.

Fernández, A. and Hill, P. (1997b). Finite domain solvers compared using self referential quizzes. In *Sixièmes Journées Francophones de Programmation Logique et Programmation par Contraintes (JFPLC'97)*, Orléans, France. LIFO. (Poster Session). Rapport de Recherche.

Fernández, A. and Hill, P. (1998a). A design for a generic constraint solver for ordered domains. In *Types for Constraint Logic Programming (TCLP'98)*, Manchester.

Fernández, A. and Hill, P. (1998b). A generic execution model for CLP(X). Research Report LCC-ITI 98/16, Universidad de Málaga, Departamento de Lenguajes y Ciencias de la Computación.

Fernández, A. and Hill, P. (1998c). An impartial efficiency comparison of FD constraints systems. In Maher, M. and Puget, J.-F., editors, *4th International Conference on Principles and Practice of Constraint Programming (CP'98)*, number 1520 in LNCS, page 468, Pisa, Italy. Springer-Verlag. Also available as Technical Report ref.98.18, School of Computer Studies, University of Leeds, September, 1998.

Fernández, A. and Hill, P. (1999a). Constrant solving on lattices. In Meo, C. and Vilares, M., editors, *International Joint Conference on Declarative Programming (APPIA-GULP-PRODE'99)*, pages 105–120, L'Aquila, Italy. Gruppo Tipografico Editoriale.

Fernández, A. and Hill, P. (1999b). Interval constraint solving over lattices using chaotic iterations. In K.Apt, Kakas, A., Monfroy, E., and Rossi, F., editors, *ERCIM/COMPULOG Workshop on Constraints*, Paphos, Cyprus. Dept., of Computer Science, University of Cyprus. Available via *http://www.cwi.nl/ERCIM/WG/Constraints/Workshops/Workshop4/Program.*

Fernández, A. and Hill, P. (1999c). An interval lattice-based constraint solving framework for lattices. In Middeldorp, A. and Sato, T., editors, *4th International Symposium on Functional and Logic Programming (FLOPS'99)*, number 1722 in LNCS, pages 194–208, Tsukuba, Japan. Springer-Verlag.

Fernández, A. and Hill, P. (2000a). A comparative study of eight constraint programming languages over the Boolean and finite domains. *Constraints*, 5(3):275–301.

Fernández, A. and Hill, P. (2000b). Constraint propagation on multiple domains. In Alpuente, M., editor, *9th International Workshop on Functional and Logic Programming (WFLP'00)*, pages 455–469, Benicássim, Spain. Universidad Politécnica de Valencia.

Fernández, A. and Hill, P. (2001a). Branching: the essence of constraint solving. In Apt, K., Barták, R., Monfroy, E., and Rossi, F., editors, *ERCIM Workshop on Constraints*, Prague, Czech Republic. Charles University/Faculty of Mathematics and Physics.

Fernández, A. and Hill, P. (2001b). A constraint system for lattice (interval) domains. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. Submitted December 2000; subjected to revision in November 2001.

Fikes, R. (1968). *A heuristic program for solving problems states as non-deterministic procedures*. PhD thesis, Comput. Sci. Dept., Carnegie-Mellon University, Pittsburgh, PA.

FLOPS (1995-96-98-99-2001). *International Symposiums*. Functional and Logic Programming. World Scientific (Singapore) and Springer-Verlag (LNCS Series).

Freeman-Benson, B. and Borning, A. (1992). Integrating constraints with an object-oriented language. In Madsen, O. L., editor, *European Conference on Object-Oriented Programming (ECOOP'92)*, number 615 in LNCS, pages 268–286, Utrecht, The Netherlands. Springer-Verlag.

Freuder, E. (1978). Synthesizing constraint expressions. *Communications of the ACM*, 21(1):958–966.

Freuder, E. and Hubbe, P. (1995). Extracting constraint satisfaction subproblems. In *14th International Joint Conference on Artificial Intelligent (IJCAI'95)*, pages 548–557, Québec, Canada. Morgan Kaufman.

Freuder, E. and Wallace, R. (1992). Partial constraint satisfaction. *Artificial Intelligence*, 58(21-70):21–70.

Frühwirth, T. (1994). Constraint handling rules. In Podelski, A., editor, *Constraint Programming: Basics and Trends*, number 910 in LNCS, pages 90–107, Châtillon-sur-Seine, France. Springer-Verlag.

Frühwirth, T. (1998). Theory and practice of constraint handling rules. *The Journal of Logic Programming*, 37:95–138.

Frühwirth, T. (1999). Compiling constraint handling rules into Prolog with attributed variables. In Nadathur, G., editor, *International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, number 1702 in LNCS, pages 117–133, Paris, France. Springer-Verlag.

Frühwirth, T. and Abdennadher, S. (2001). The Munich rent advisor: a success for logic programming on the internet. *Theory and Practice of Logic Programming*, 1(3):303–319.

Frühwirth, T. and Brisset, P. (1997). Optimal planning of digital cordless telecommunication systems. In Wallace, M., editor, *3rd International Conference on the Practical Application of Constraint Technology (PACT'97)*, pages 165–176, London, UK. Prolog Management Group.

Frühwirth, T. and Brisset, P. (1998). Optimal placement of base stations in wireless indoor telecommunication. In Maher, M. and Puget, J.-F., editors, *4th International Conference on Principles and Practice of Constraint Programming (CP'98)*, number 1520 in LNCS, pages 476–480, Pisa, Italy. Springer-Verlag.

Frühwirth, T., Herold, A., Kuechenhoff, V., Le Provost, T., Lim, P., Monfroy, E., and Wallace, M. (1993). Constraint logic programming - an informal introduction. In Comyn, G., Ratcliffe, M., and Fuchs, N., editors, *2nd International Logic Programming Summer School (LPSS'92): Logic programming in Action*, number 636 in LNAI, Zurich, Switzerland. Springer-Verlag.

Gaschnig, J. (1974). A constraint satisfaction method for inference making. In *12th Annual Allerton Conference on Circuit System Theory*, pages 866–874, Illinois University.

Georget, Y. and Codognet, P. (1998). Compiling semiring-based constraints with clp(FD,S). In Maher, M. and Puget, J.-F., editors, *4th International Conference on Principles and Practice of Constraint Programming (CP'98)*, number 1520 in LNCS, pages 205–219, Pisa, Italy. Springer-Verlag.

Gervet, C. (1994). Conjunto: constraint logic programming with finite set domains. In Bruynooghe, M., editor, *1994 International Symposium on Logic programming (SLP'94)*, pages 339–358, Ithaca, New York. The MIT Press.

Gervet, C. (1997). Interval propagation to reason about sets: definition and implementation of a practical language. *Constraints*, 1(3):191–244.

Gilbert, D., Schroeder, M., and van Helden, J. (2000). Interactive visualisation and exploration of biological data. In Levi, G. and Martelli, M., editors, *5th Joint Conference on Information Sciences (Stream on Biomolecular Informatics)*, Atlantic City, New Jersey, USA.

Gorlick, M., Kesselman, C., Marotta, D., and Parker, D. (1990). Mockingbird: a logical methodology for testing. *Journal of Logic Programming*, 8(1):95–119.

Goualard, F. (2001). Component programming and interoperatibility in constraint solver design. In K.Apt, Barták, R., Monfroy, E., and Rossi, F., editors, *ERCIM Workshop on Constraints*, Prague, Czech Republic. Charles University/Faculty of Mathematics and Physics.

Goualard, F., Benhamou, F., and Granvilliers, L. (1999). An extension of the WAM for hybrid interval solvers. *The Journal of Functional and Logic Programming*, 1999(1):1–36. Special issue of Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages.

Granvilliers, L. (2001). On the combination of interval constraint solvers. *Reliable Computing*, 7(6):467–483.

Granvilliers, L., Monfroy, E., and Benhamou, F. (2001). Cooperative solvers in constraint programming: a short introduction. *ALP Newsletter*, 14(2).

Hansen, E. (1992). *Global optimization using interval analysis.* Marcel Dekker.

Hansen, P., Jaumard, B., and Mathon, V. (1993). Constrained nonlinear 0-1 programming. *Journal on Computing*, 5(2):97–119.

Hanus, M. (1993). Analysis of nonlinear constraints in CLP($\Re$). In Warren, D., editor, *10th International Conference on Logic Programming (ICLP'93)*, pages 83–99, Budapest, Hungary. The MIT Press.

Haralick, R. and Elliot, G. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313.

Havens, W., Sidebottom, S., Sidebottom, G., Jones, J., and Ovans, R. (1992). Echidna: a constraint logic programming shell. In *Pacific Rim International Conference on Artificial Intelligence*, pages 165–171, Seoul, Korea.

Heintze, N., Michaylov, S., and Stuckey, P. (1992). CLP($\Re$) and some electrical engineering problems. *Journal of Automated Reasoning*, 9(2):231–260.

Heintze, N., Michaylov, S., Stuckey, P., and Yap, R. (1989). On meta-programming in CLP($\Re$). In Lusk, E. and Overbeek, R., editors, *the 1989 North American Conference on Logic Programming (NACLP'89)*, pages 52–66, Cleveland, Ohio. The MIT Press.

Henz, M. (1996). Don't be puzzled! In *Workshop on Constraint Programming in conjunction with the 2nd International Conference on Principles and Practice of Constraint Programming (CP'96)*, Cambridge, Massachusetts, USA.

Henz, M. and Müller, T. (2000). An overview of finite domain constraint programming. In *5th Conference of the Association of Asia-Pacific Operational Research Societies*, Singapore. to appear.

Hermenegildo, M., Bueno, F., Cabeza, D., Carro, M., de la Banda, M., López-García, P., and Puebla, G. (2000). The Ciao logic programming environment. In Lloyd, J. W., Dahl, V., Furbach, U., Kerber, M., Lau, K., Palamidessi, C., Pereira, L., Sagiv, Y., and Stuckey, P. J., editors, *1st International Conference on Computational Logic (CL'2000)*, number 1861 in LNCS, London, UK. Springer-Verlag. Tutorial.

Hickey, T. (2000). CLIP: a CLP(Intervals) dialect for metalevel constraint solving. In Pontelli, E. and Costa, V., editors, *2nd International Workshop on Practical Aspects of Declarative Languages (PADL'2000)*, number 1753 in LNCS, pages 200–214, Boston, USA. Springer-Verlag.

Hickey, T., Ju, Q., and van Emden, M. (1999). Interval arithmetic: from principles to implementation. CS Technical report CS-99-202, Brandeis Iniversity.

Hofstedt, P. (2000). Better communication for tighter cooperation. In Lloyd, J. W., Dahl, V., Furbach, U., Kerber, M., Lau, K., Palamidessi, C., Pereira, L., Sagiv, Y., and Stuckey, P. J., editors, *1st International Conference on Computational Logic (CL'2000)*, number 1861 in LNCS, pages 342–357, London, UK. Springer-Verlag.

Holzbaur, C. (1995). OFAI clp(q,r) manual, edition 1.3.3. Technical Report TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna.

Hong, H. (1993). RISC-CLP(Real): logic programming with non-linear constraint over reals. In *(Benhamou and Colmerauer, 1993)*, pages 133–159, Cambridge, MA. The MIT Press.

Hong, H. and Ratschan, S. (1995). RISC-CLP(Tree(Δ)) a constraint logic programming system with parametric domain. Technical Report 95-25, Research Institute for Symbolic Computation, Johannes Kepler University, A-4040 Linz, Austria.

Hudson, D. and Cohen, M. (2000). *Neural networks and artificial intelligence for biomedical engineering*. Biomedical Engineering. The IEEE Press, New York, USA.

Hughes, J. (1989). Why functional programming matters. *Computer Journal*, 32(2):98–107.

Huynh, T. and Lassez, C. (1988). A CLP($\Re$) options trading analysis system. In Kowalski, R. and Bowen, K., editors, *5th International Conference and Symposium of Logic Programming (ICLP/SLP'88)*, pages 59–69, Seattle, Washington. The MIT Press.

Hyvönen, E. (1989). Constraint reasoning based on interval arithmetic. In Sridharan, N. S., editor, *11th International Joint Conference on Artificial Intelligent (IJCAI'89)*, pages 1193–1198, Detroit, MI, USA. Morgan Kaufman.

If/Prolog (1994). *IF/Prolog V5.0A, constraints package*. Siemens Nixdorf Informationssysteme AG, Munich, Germany.

Ilog (1995). Ilog SOLVER, reference manual, version 3.1.

Jaffar, J. and Lassez, J. (1987). Constraint logic programming. In *14th ACM Symposium on Principles of Programming Languages (POPL'87)*, pages 111–119, Munich, Germany. ACM Press.

Jaffar, J. and Maher, M. (1994). Constraint logic programming: a survey. *The Journal of Logic Programming*, 19-20:503–581.

Jaffar, J., Maher, M., Marriot, K., and Stuckey, P. (1998). The semantics of constraint logic programs. *The Journal of Logic Programming*, 37:1–46.

Jaffar, J. and Michaylov, S. (1987). Methodology and implementation of a CLP system. In Lassez, J.-L., editor, *4th International Conference on Logic Programming (ICLP'87)*, pages 196–218, Melbourne, Australia. The MIT Press.

Jaffar, J., Michaylov, S., Stuckey, P., and Yap, R. (1992a). An abstract machine for CLP($\Re$). *SIGPLAN Notices*, 27(7):128–139. Publication of the Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI'92), San Francisco, California.

Jaffar, J., Michaylov, S., Stuckey, P., and Yap, R. (1992b). The CLP($\Re$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395.

Jaffar, J., Michaylov, S., and Yap, R. (1991). A methodology for managing hard constraints in CLP systems. *SIGPLAN Notices*, 26(6):306–316. Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI'91), Toronto, Ontario, Canada.

Jampel, M. (1994). A review of "A review of industrial constraint solving tools". It is a review of (Cras, 1993). Available at *http://www.cs.unh.edu/ccc/archive/constraints/archive/cras.html*.

Janson, S. and Haridi, S. (1991). Programming paradigms of the Andorra kernel language. In Saraswat, V. and Ueda, K., editors, *1991 International Symposium on Logic Programming (ISLP'91)*, Logic Programming, pages 167–183, San Dieg, California, USA. The MIT Press.

JFLP (1995-2000). Journal of Functional and Logic Programming. Volumes 1995-2000 published by the MIT Press. From 2001 published by the European Association for Programming Languages and Systems (EAPLS).

Kepser, S. and Richts, J. (1999). Optimisation techniques for combining constraint solvers. In Gabbay, D. and de Rijke, M., editors, *Frontiers of Combining Systems 2 (FroCoS'98)*, pages 193–210, Amsterdam. Research Studies Press/Wiley.

King, A. (2000). Pair-sharing over rational trees. *Journal of Logic Programming*, 46(1-2):139–155.

Kok, J., Marchiori, E., Marchiori, M., and Rossi, C. (1996). Evolutionary training of CLP-constrained neural networks. In Wallace, M., editor, *2nd International Conference on the Practical Application of Constraint Technology (PACT'96)*, pages 129–142, London, UK. Publisher Prolog Management Group.

Kondrak, G. and Van Beek, P. (1997). A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 89(1-2):365–387.

Kowalski, R. (1974). Predicate logic as programming language. In Rosenfeld, J., editor, *Information Processing, IFIP Congress 74*, pages 569–574, Stockholm, Sweden. North-Holland.

Kowalski, R. (1988). The early years of logic programming. *Communications of the ACM*, 31(1):38–43.

Kowalski, R. and Kuehner, D. (1971). Linear resolution with selection function. *Artificial Intelligence*, 2(3-4):227–260.

Kozen., D. (1994). Set constraints and logic programming. In Jouannaud, J.-P., editor, *1st International Conference on Constraints in Computational Logics (CCL'94)*, number 845 in LNCS, pages 302–303, Munich, Germany. Springer-Verlag. Extended version published in (Kozen., 1998).

Kozen., D. (1998). Set constraints and logic programming. *Information and Computation*, 142(1):2–25.

Kumar, V. (1992). Algorithms for constraint satisfaction problems: a survey. *AI Magazine*, 13(1):32–44.

Lassez, C. (1987). Constraint logic programming: a tutorial. *BYTE magazine*, pages 171–176.

Le Provost, T. and Wallace, M. (1993). Generalized constraint propagation over the CLP scheme. *Journal of Logic Programming*, 16(3):319–359.

Lee, J. and van Emden, M. (1993). Interval computation as deduction in CHIP. *The Journal of Logic Programming, Special Issue:Constraint Logic Programming*, 16(3-4):255–276.

Legeard, B. and Legros, E. (1991). Short overview of the CLPS system. In Maluszynski, J. and Wirsing, M., editors, *3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP'97)*, number 528 in LNCS, pages 431–433, Passau, Germany. Springer-Verlag.

Levi, G., editor (1994). *Advances in logic programming theory*, volume 1. Oxford University Press, UK.

Lloyd, J. (1987). *Foundations of logic programming.* Springer-Verlag, Berlin, Heidelberg.

Lux, W. (2001). Adding linear constraints over real numbers to Curry. In Middeldorp, A., Kuchen, H., and Ueda, K., editors, *5th International Symposium on Functional and Logic Programming (FLOPS'2001)*, number 2024 in LNCS, pages 185–200, Tokyo, Japan. Springer-Verlag.

Mackworth, A. (1977). Consistency in networks of relations. *Artificial Intelligence*, 8:99–118.

Maher, M. (1987). Logic semantics for a class of committed-choice programs. In Lassez, J.-L., editor, *4th International Conference on Logic Programming (ICLP'87)*, pages 858–876, Melbourne, Australia. The MIT Press.

Majumdar, S. (1997). Application of relational interval arithmetic to computer performance analysis: a survey. *Constraints*, 2(2):215–235.

Maluszynski, J. and Wirsing, M., editors (1991). *3rd International Symposium in Programming Language Implementation and Logic Programming (PLILP'91)*, volume LNCS 528. Springer-Verlag, Passau, Germany.

Marriot, K. and Stuckey, P. J. (1998). *Programming with constraints.* The MIT Press, Cambridge, Massachusetts.

Meseguer, P. and Larrosa, J. (1995). Constraint satisfaction as global optimization. In *14th International Joint Conference on Artificial Intelligent (IJCAI'95)*, pages 579–585, Québec, Canada. Morgan Kaufman.

Mohr, R. and Henderson, T. (1986). Arc and path consistency revisited. *Artificial Intelligence*, 28(2):225–233.

Monfroy, E. (1996). *Solver collaboration for constraint logic programming.* PhD thesis, Centre de Recherche en Informatique de Nancy, INRIA-Lorraine.

Monfroy, E., Rusinowitch, M., and Schott, R. (1995). Implementing non-linear constraints with cooperative solvers. Research Report 2747, Centre de Recherche en Informatique de Nancy, INRIA-Lorraine.

Montanari, U. and Rossi, F. (1991). Constraint relaxation may be perfect. *Artificial Intelligence*, 48(2):143–170.

Moore, R. (1966). *Interval analysis.* Prentice hall, Englewood Cliffs, NJ.

Müller, T. and Müller, M. (1997). Finite set constraints in Oz. In Bry, F., Freitag, B., and Seipel, D., editors, *13th Workshop on Logic Programming*, pages 104–115, München. Technische Universität.

Müller, T. and Würtz, J. (1996). Interfacing propagators with a concurrent constraint language. In *JICSLP96 Post-conference workshop and Compulog Net Meeting on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, pages 195–206, Bonn, Germany.

Nadel, B. (1989). Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224.

N'Dong, S. (1997). Prolog IV ou la programmation par contraintes selon PrologIA. In *Sixièmes Journées Francophones de Programmation Logique et Programmation par Contraintes (JFPLC'97)*, pages 235–238, Orléans, France. Edition HERMES.

Older, W. (1989). Interval arithmetic specification. Technical report, Bell-Northern, Research Computing Research Laboratory, Ottawa, Ontario, Canada.

Older, W. and Benhamou, F. (1993). Programming in CLP(BNR). 1st International Workshop on Principles and Practice of Constraint Programming (PPCP'93), Informal Proceedings, pages: 228-238, Brown University, Newport, Rode Island.

Older, W. and Vellino, A. (1990). Extending Prolog with constraint arithmetic on real intervals. In *The Canadian Conference on Computer and Electrical Engineer*, Ottawa, Ontario, Canada. IEEE.

Older, W. and Vellino, A. (1993). Constraint arithmetic on real intervals. In *(Benhamou and Colmerauer, 1993)*, pages 175–195, Cambridge, MA. The MIT Press.

Pachet, F. and Roy, P. (1995). Integrating constraint satisfaction techniques with complex object structures. In *Annual Conference on the British Computer Society on Expert Systems (ES'95)*, pages 11–22. Cambridge.

PACLP'2000 (2000). *2nd International Conference of The Practical Applications of Constraint Technology and Logic Programming*. Practical Application Company, Manchester, UK.

PACLP'99 (1999). *1st International Conference of The Practical Applications of Constraint Technology and Logic Programming*. Practical Application Company, Manchester, UK.

PACT'96 (1996). *2nd International Conference of Practical Application of Constraint Technology*. Prolog Management Group, London, UK.

PACT'97 (1997). *3rd International Conference of Practical Application of Constraint Technology*. Prolog Management Group, London, UK.

PAPPACT'98 (1998). *6th International Conference of Practical Application of Prolog and the 4th International Conference on the Practical Application of Constraint Technology*. Publisher Practical Application Company Ltd, London, UK.

Pesant, G. and Boyer, M. (1994). QUAD-CLP($\Re$): adding the power of quadratic constraints. In Borning, A., editor, *2nd International Workshop on Principles and Practice of Constraint Programming (PPCP'94)*, number 874 in LNCS, pages 95–108, Orcas Island, Washington, USA. Springer-Verlag.

Prosser, P. (1993). Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299.

Puget, J.-F. and Leconte, M. (1995). Beyond the glass box: constraints as objects. In J. W. Lloyd, editor, *International Symposium on Logic Programming (ILPS'95)*, pages 513–527, Portland, Oregon. The MIT Press.

Refalo, P. and Van Hentenryck, P. (1996). CLP($\Re_{lin}$) revised. In Maher, M., editor, *Joint International Conference and Symposium on Logic Programming (JICSLP'96)*, pages 22–36, Bonn, Germany. The MIT Press.

Régin, J.-C. (1994). A filtering algorithm for constraints of difference in CSPs. In Levi, G. and Martelli, M., editors, *12th National Conference on Artificial Intelligence*, volume 1, pages 362–367, Seattle,WA,USA. AAAI Press.

Robinson, A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41.

Robinson, A. (1971). Computational logic: The unification computation. *Machine Intelligence*, 6:63–72.

Ruttkay, Z. (1998). Constraint satisfaction-a survey. *CWI Quaterly*, 11(2-3):163–214.

Sakai, K. and Aiba, A. (1989). CAL: a theoretical background of constraint logic programming and its applications. *Journal of Symbolic Computations*, 8(6):589–603.

Saraswat, V. (1988). A somewhat logical formulation of CLP synchronisation primitives. In Kowalski, R. and Bowen, K., editors, *5th International Conference and Symposium of Logic Programming (ICLP/SLP'88)*, pages 1298–1314, Seattle, Washington. The MIT Press.

Saraswat, V. (1989). *Concurrent constraint programming languages.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA. Also published in (Saraswat, 1993).

Saraswat, V. (1992). Concurrent constraint programming: A brief survey. Unpublished.

Saraswat, V. (1993). *Concurrent constraint programming.* The MIT Press, Cambridge, MA. Doctoral Dissertation Award and Logic programming Series.

Sato, S. and Aiba, A. (1993a). An application of CAL to robotics. In *(Benhamou and Colmerauer, 1993)*, pages 161–173, Cambridge, MA. The MIT Press.

Sato, S. and Aiba, A. (1993b). A study on Boolean constraint solvers. In *(Benhamou and Colmerauer, 1993)*, pages 253–267, Cambridge, MA. The MIT Press.

Schiex, T., Fargier, H., and Verfaillie, G. (1995). Valued constraint satisfaction problems: hard and easy problems. In *14th International Joint Conference on Artificial Intelligent (IJCAI'95)*, pages 631–637, Québec, Canada. Morgan Kaufman.

Schulte, C. (1995). Solver– an Oz search debugger. In *International Workshop on Oz Programming (WOz'95)*, pages 109–115, Martigny, Switzerland. Institut Dalle Molle d'Intelligence Artificielle Perceptive!

Sedgewick, R. (1984). *Algorithms*. Series in Computer Science. Addison-Wesley, USA.

Shapiro, E. (1989). The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510.

Sicstus manual (1994). *SICStus Prolog user's manual, release* 3#5. By the Intelligent Systems Laboratory, Swedish Institute of Computer Science.

Sidebottom, G. (1993). *A language for optimizing constraint propagation*. PhD thesis, Simon Fraser University, Burnaby, Canada.

Sidebottom, G. and Havens, W. (1992). Hierarchical arc consistency for disjoint real intervals in constraint logic programming. *Computational Intelligence*, 8(4):601–623.

Simonis, H. (1995). Applications of constraint logic programming. In Sterling, L., editor, *12th International Conference on Logic Programming (ICLP'95)*, pages 9–11, Tokyo, Japan. The MIT Press. Advanced Tutorials.

Simonis, H. and Dincbas, M. (1987). Using logic programming for fault diagnosis in digital circuits. In Morik, K., editor, *11th German Workshop on Artificial Intelligence (GWAI'87)*, pages 139–148, Geseke. Springer-Verlag.

Smith, B. (1995). A tutorial on constraint programming. Research Report 95.14, University of Leeds, School of Computer Studies, England.

Smolka, G. (1995). The Oz programming model. In Van Leeuwen, J., editor, *Computer Science Today*, number 1000 in LNCS, pages 324–343, Berlin. Springer-Verlag.

Sterling, L. and Shapiro, E. (1986). *The art of Prolog*. Series in Logic Programming. The MIT Press, Cambridge, MA.

Stolzenburg, F. (1996). Membership-constraints and complexity in logic programming with sets. In Baader, F. and Schulz, K., editors, *1st International Workshop on Frontiers of Combining Systems (FroCos'96)*, volume 3 of *Applied Logic*, pages 285–302, Munich, Germany. Kluwer Academic.

Tsang, E. (1993). *Foundations of constraint satisfaction.* Academic Press, London and San Diego.

Van Emdem, M. (1997). Value constraints in the CLP scheme. *Constraints*, 2(2):163–183.

Van Hentenryck, P. (1988). Tutorial on the CHIP system and applications. In *Workshop of Constraint Logic Programming*, Rehovot, Israel. Weizmann Institute of Science.

Van Hentenryck, P. (1989). *Constraint satisfaction in logic programming.* The MIT Press, Cambridge, MA.

Van Hentenryck, P. (1995). Constraint solving for combinatorial search problems: a tutorial. In Montanari, U. and Rossi, F., editors, *1st International Conference on Principles and Practice of Constraint Programming (CP'95)*, number 976 in LNCS, pages 564–587, Cassis, France. Springer-Verlag.

Van Hentenryck, P. (1998). A gentle introduction to NUMERICA. *Artificial Intelligence*, 103(1-2):209–235.

Van Hentenryck, P. (1999). *The OPL optimization programming language.* The MIT Press, Cambridge, MA.

Van Hentenryck, P. and Deville, Y. (1991). The cardinality operator: a new logical connective for constraint logic programming. In Furukawa, K., editor, *8th International Conference on Logic Programming (ICLP'91)*, pages 745–759, Paris, France. The MIT Press.

Van Hentenryck, P., Deville, Y., and Teng, C. (1992). A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2-3):291–321.

Van Hentenryck, P., Michel, L., and Benhamou, F. (1988). Newton - constraint programming over nonlinear constraints. *Science of Computer Programming*, 20(1-2):83–118.

Van Hentenryck, P., Michel, L., and Deville, Y. (1997). *Numerica: a modeling language for global optimization.* The MIT Press, Cambridge, MA.

Van Hentenryck, P., Michel, L., Perron, L., and Régin, J.-C. (1999). Constraint programming in OPL. In *International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, number 1702 in LNCS, pages 98–116, Paris, France. Springer-Verlag.

Van Hentenryck, P., Saraswat, V., and Deville, Y. (1991). Constraint processing in cc(FD). Unpublished draft.

Van Hentenryck, P., Saraswat, V., and Deville, Y. (1994). Design, implementation and evaluation of the constraint language cc(FD). In Podelski, A., editor, *Constraint*

Programming: Basics and Trends, number 910 in LNCS, pages 293–316, Châtillon-sur-Seine, France. Springer-Verlag.

van Hoeve, W. (2001). The "all different" constraint: a survey. In K.Apt, Barták, R., Monfroy, E., and Rossi, F., editors, *ERCIM Workshop on Constraints*, Prague, Czech Republic. Charles University/Faculty of Mathematics and Physics.

Walinsky, C. (1989). CLP(Σ^*): constraint logic programming with regular sets. In Levi, G. and Martelli, M., editors, *6th International Conference on Logic Programming (ICLP'89)*, pages 181–196, Lisbon, Portugal. The MIT Press.

Wallace, M. (1996). Practical applications of constraint programming. *Constraints*, 1(1-2):139–168.

Wallace, R. (1993). Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs. In Bajcsy, R., editor, *13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 239–247, Chambéry, France. Morgan Kaufmann.

Waltz, D. (1972). Generating semantic descriptions from drawings of scenes with shadows. Technical Report AI271, MIT, MA.

Waltz, D. (1975). Understanding line drawings in scenes shadows. In Winston, P. H., editor, *The Psychology of Computer Vision*, pages 19–91, UK. McGraw-Hill.

Warren, D. (1983). An abstract Prolog instruction set. technical Note 309, SRI International, Menlo Park.

Williams, H. (1993a). *Model building in mathematical programming.* J. Wiley and Sons, New York, USA. Revised edition.

Williams, H. (1993b). *Model solving in mathematical programming.* J. Wiley and Sons, New York, USA.

Wirth, N. (1966). A contribution to the development of Algol. *Communications of the ACM*, 9(6):413–431.

Yap, R. (1991). Restriction site mapping in CLP($\Re$). In Furukawa, K., editor, *8th International Conference on Logic Programming (ICLP'91)*, pages 521–534, Paris, France. The MIT Press.

Yap, R. (1993). A constraint logic programming framework for constructing DNA restriction maps. *Artificial Intelligence in Medicine*, 5:447–464.

Zhou, J. (2000a). Introduction to the constraint language NCL. *The Journal of Logic Programming*, 45(1-3):71–103.

Zhou, N.-F. (1994). Parameter passing and control stack management in Prolog implementation revisited. *ACM Transactions on Programming Languages and Systems*, 18(6):752–779.

Zhou, N.-F. (1996). A high-level intermediate language and the algorithms for compiling finite-domain constraints. In Jaffar, J., editor, *Joint International Conference and Symposium on Logic Programming (JICSLP'98)*, pages 70–84, Manchester, UK. The MIT Press.

Zhou, N.-F. (1997). B-Prolog user's manual (version 2.1). Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology, Fukuoka, Japan.

Zhou, N.-F. (2000b). B-Prolog user's manual (version 4.0). Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology, Fukuoka, Japan.

Zhou, N.-F. and Nagasawa, I. (1994). An efficient finite-domain constraint solver in beta-Prolog. *Journal of Japanese Society for Artificial Intelligence*, 9:275–282.

Zhou, N.-F., Takagi, T., and Ushijima, K. (1990). A matching tree oriented abstract machine for Prolog. In Warren, D. and Szeredi, P., editors, *7th International Conference on Logic Programming (ICLP'90)*, pages 159–173, Jerusalem, Israel. The MIT Press.

Appendix A

Computation Domains in $clp(\mathcal{L})$

The following code is extracted directly from the file clp_l.pl that is provided with the distribution of the prototype implementation of our generic solver (Fernández, 2000).

Declarations of simple computation domains

```
%-------------- Integer Domain ---------------------------------------
%--- Definition
lattice(Ele,integer):-(integer(Ele);var(Ele)),!.

%--- Cardinality
finite(integer).

%--- For finite domains  ----Successor AND Predecessor
succ(integer,X,Y):-lattice(X,integer),
                   lattice(Y,integer),
                   Y is X+1.

pred(integer,X,Y):-lattice(X,integer),
                   lattice(Y,integer),
                   Y is X-1.

%--- Ordering
lt(normal,integer,A,B):-lattice(A,integer),lattice(B,integer),!,A < B.

%--- Glb and Lub
glb(normal,integer,X,Y,Z):-lattice(X,integer),lattice(Y,integer),
                           (lt(normal,integer,X,Y) -> Z=X;Z=Y).

lub(normal,integer,X,Y,Z):- lattice(X,integer),lattice(Y,integer),
                           (gt(normal,integer,X,Y) -> Z=X;Z=Y).
```

```
%--- Top and bottom elements
%% They are fictitious.

%-------------- Real Domain ------------------------------------------
%--- Definition
lattice(Ele,real):-(float(Ele);var(Ele)),!.

%--- Cardinality
finite(real):-fail,!.

%--- Ordering
lt(normal,real,A,B):- lattice(A,real),lattice(B,real),!,A < B.

%--- Glb and Lub
glb(normal,real,X,Y,Z):- lattice(X,real),lattice(Y,real),
                         (lt(normal,real,X,Y) -> Z=X;Z=Y).

lub(normal,real,X,Y,Z):- lattice(X,real),lattice(Y,real),
                         (gt(normal,real,X,Y) -> Z=X;Z=Y).

%--- Top and bottom elements
%% They are fictitious.

%--- Precision
%% See below

%-------------- Set Domain -------------------------------------------
%--- Definition
% A set is defined as a list with ordering the inclusion.
:-use_module(library(lists)).
:-dynamic set/1.

lattice(Ele,set):-is_list(Ele),!.

%--- Cardinality of set domain
finite(set):-fail.

%--- Set Ordering.
include_in([],_S).
include_in([X|Y],S):-!,member(X,S),include_in(Y,S).

lt(normal,set,S1,S2):-lattice(S1,set),lattice(S2,set),
                      S1\==S2,include_in(S1,S2).
```

```
%--- Glb and Lub
union(S1,S2,S3):-lattice(S1,set),lattice(S2,set),!,
                 append(S1,S2,S),remove_duplicates(S,S3).

intersection([],S2,[]):-lattice(S2,set),!.
intersection([X|Y],S2,[X|S]):-lattice(S2,set),member(X,S2),
                        !,intersection(Y,S2,S).
intersection([X|Y],S2,S):-lattice(S2,set),not(member(X,S2)),
                        !,intersection(Y,S2,S).

glb(normal,set,X,Y,Z):- lattice(X,set),lattice(Y,set),
                        !, intersection(X,Y,Z).
lub(normal,set,X,Y,Z):- lattice(X,set),lattice(Y,set),
                        !,union(X,Y,Z).

%--- Top element is fictitious
 bottom(set,[]).
 top(set,top).

%-------------- Colors Domain ----------------------------------------
:-dynamic color/1.

color(white).
color(yellow).
color(orange).
color(green).
color(blue).
color(brown).
color(black).

lattice(Ele,colors):- (color(Ele);var(Ele)),!.

%--- Cardinality
finite(colors).

%--- For finite domains, we declare the successor and predecessor
succ(colors,white,yellow).
succ(colors,yellow,orange).
succ(colors,orange,green).
succ(colors,green,blue).
succ(colors,blue,brown).
succ(colors,brown,black):-!.
succ(colors,black,black).                %% succ(colors,top,top).
```

```
pred(colors,white,white).                %% pred(colors,bottom,bottom).
pred(colors,X,Y):-succ(colors,Y,X),!.

%--- Ordering
lt(normal,colors,A,A):-lattice(A,colors),!,fail.
lt(normal,colors,X,Y):-lattice(X,colors),lattice(Y,colors),
                       succ(colors,X,Y),!.
lt(normal,colors,X,Y):-lattice(X,colors),lattice(Y,colors),
                       succ(colors,X,Z),
                       X\==Z,!,lt(normal,colors,Z,Y).

%--- Glb and Lub
glb(normal,colors,X,Y,Z):- lattice(X,colors),lattice(Y,colors),!,
                           (lt(normal,colors,X,Y) -> Z=X;Z=Y).

lub(normal,colors,X,Y,Z):- lattice(X,colors),lattice(Y,colors), !,
                           (gt(normal,colors,X,Y) -> Z=X;Z=Y).

%--- Top and bottom elements.
bottom(colors,white).
top(colors,black).
```

An example of precision declaration

The following code declares a precision of 0.05 for the real domain. aplica/1 is a built-in predicate that, given a list with three elements, applies a binary operator to the two first elements (i.e., the operands) that belong to some domain L^s (and, thus, have the form (R,B) where R belongs to some domain and B is either 'open' or 'close') and puts the result in the last element of the list (i.e., Prec). This last element is used to check the precision. In the example below, if Prec contains a value lower or equal to the value *Epsilon*] in $\Re^s$, the precision predicate returns true indicating that the system has reached the precision expected for the real domain (of course, Epsilon may be provided by the user).

```
%--- Precision
precision((R1,B1),(R2,B2),real,Epsilon):-lattice(B1,bracket),
                         lattice(B2,bracket),
                         lattice(R1,real), lattice(R2,real),
                         !,
                         aplica(:-:,[(R2,B2),(R1,B1),Prec]),
                         le(real,Prec,(Epsilon,close)).
```

Examples of combined domains

```
%----------------------DIRECT PRODUCT ---------------------------------
%-------------------  POINT DOMAIN -----------------------------------
% Integer Point domain is the direct product of Integer x Integer
% Real Point domain is the direct product of Real x real
% Integer-Real Point domain is the direct product of Integer x Real
% Real-Integer Point domain is the direct product of Real x Integer

product_Direct(integer,integer,int_point).
product_Direct(real,real,real_point).
product_Direct(integer,real,intre_point).
product_Direct(real,integer,reint_point).

%------------------- A RECTANGLE DOMAIN -------------------------------
product_Direct(int_point,int_point,rectangle).
```

Appendix B

Constraint Operators in $clp(\mathcal{L})$

Some simple operators

```
%---------------- DECLARATION OF UNARY AND BINARY OPERATORS -----------
% Each operator must be declared as dynamic and with a priority 625.
% The definition of an operator is done for the bracket
% domain as well as the computation domain.
%-------------------- Sum operator ----------------------------------
:-dynamic ':+:'/3.
operator(625,xfx,':+:').

%--- Mode declaration. L2 is not defined as L1 to allow
%    the definition of the sum on combined domain (e.g. '1':+:'1'='10'
%    in the hexadecimal domain)
declara(:+:,L1,L1,_L2).       %% ':+:' :: L1 x L1 -> L2

%--- Def. on Bracket domain
:+:(open,_,open).
:+:(close,B,B).

%--- Def. on integer domain.
:+:(Ele1,Ele2,Ele3):-lattice(Ele1,integer), lattice(Ele2,integer),
                     Ele3 is  Ele1 + Ele2.

%--- Def. on real domain.
:+:(Ele1,Ele2,Ele3):-lattice(Ele1,real), lattice(Ele2,real),
                     Ele3 is  Ele1 + Ele2.

%--- Def. on the integer point domain.
:+:((A,B),(C,D),(E,F)):-lattice((A,B),int_point),
                        lattice((C,D),int_point),
```

```
                        E is  A + C, F is B + D.

%--- Def. on the real point domain.
:+:((A,B),(C,D),(E,F)):-lattice((A,B),real_point),
                        lattice((C,D),real_point),
                        E is  A + C, F is B + D.

%--- Def. on the integer-real point domain.
:+:((A,B),(C,D),(E,F)):-lattice((A,B),intre_point),
                        lattice((C,D),intre_point),
                        E is  A + C, F is B + D.

%--- Def. on the real-integer point domain.
:+:((A,B),(C,D),(E,F)):-lattice((A,B),reint_point),
                        lattice((C,D),reint_point),
                        E is  A + C, F is B + D.

%--- Def. on the set domain.
:+:(Ele1,Ele2,Ele3):-lattice(Ele1,set), lattice(Ele2,set),
                      union(Ele1,Ele2,Ele3).

%-------------------- Subtraction operator ----------------------------
:-dynamic ':-:'/3.
operator(625,xfx,':-:').

%--- Mode declaration. L2 is not defined as L1 to allow
% the definition of the subtraction on combined domain
%% (e.g. '10':-:'F'='1' taking into account the  hexadecimal domain)
declara(:-:,_L2,mirror(L1),L1).     %% ':-:':: L2 x L1-> L1

%--- Subtraction on bracket domain
:-:(A,A,close).
:-:(A,B,open):-A\==B.

%--- Subtraction on Integers
:-:(Ele1,Ele2,Ele3):-lattice(Ele1,integer),lattice(Ele2,integer),
                     Ele3 is  Ele1 - Ele2.

%--- Subtraction on Reals
:-:(Ele1,Ele2,Ele3):-lattice(Ele1,real),lattice(Ele2,real),
                     Ele3 is  Ele1 - Ele2.

%--- Def. on the integer point domain.
:-:((A,B),(C,D),(E,F)):-lattice((A,B),int_point),
```

```
                        lattice((C,D),int_point),
                        E is  A - C, F is B - D.

%--- Def. on the real point domain.
:-:((A,B),(C,D),(E,F)):-lattice((A,B),real_point),
                        lattice((C,D),real_point),
                        E is  A - C, F is B - D.

%--- Def. on the integer-real point domain.
:-:((A,B),(C,D),(E,F)):-lattice((A,B),intre_point),
                        lattice((C,D),intre_point),
                        E is  A - C, F is B - D.

%--- Def. on the real-integer point domain.
:-:((A,B),(C,D),(E,F)):-lattice((A,B),reint_point),
                        lattice((C,D),reint_point),
                        E is  A - C, F is B - D.

%--- Def. on the set domain.
:-:(Ele1,Ele2,Ele3):-lattice(Ele1,set), lattice(Ele2,set),
                     difference(Ele1,Ele2,Ele3).

%------------------- Product operator --------------------------------
:-dynamic ':*:'/3.
operator(625,xfx,':*:').

%--- Mode declaration
declara(:*:,L,L,L).       %% ':*:' :: L x L -> L

%--- Def. on Bracket domain
:*:(open,_,open).
:*:(close,B,B).

%--- Def. on integer domain.
:*:(Ele1,Ele2,Ele3):-lattice(Ele1,integer), lattice(Ele2,integer),
                     Ele3 is  Ele1 * Ele2.

%--- Def. on real domain.
:*:(Ele1,Ele2,Ele3):-lattice(Ele1,real), lattice(Ele2,real),
                     Ele3 is  Ele1 * Ele2.

%------------------- Division operator -------------------------------
:-dynamic ':/:'/3.
operator(625,xfx,':/:').
```

```
%--- Mode declaration                                  _
declara(:/:,L,mirror(L),L).       %% ':/:' :: L x L  -> L

%--- Def. on Bracket domain
:/:(open,_,open).
:/:(close,B,B).

%--- Def. on integer domain.
:/:(Ele1,Ele2,Ele3):-lattice(Ele1,integer), lattice(Ele2,integer),
                     Ele2 \== 0,   %% To avoid division by zero
                     Ele3 is  integer(Ele1/Ele2).

%--- Def. on real domain.
:/:(Ele1,Ele2,Ele3):-lattice(Ele1,real), lattice(Ele2,real),
                     Ele2 \== 0,    %% To avoid division by zero
                     Ele3 is  Ele1/Ele2.

%------------------- Binary Sqrt operator -----------------------------
:-dynamic ':&:'/3.
operator(625,xfx,':&:').

%--- Mode declaration
declara(:&:,L,L,L).       %% ':&:' :: L x L -> L

%--- Def. on Bracket domain
:&:(open,_,open).
:&:(close,_B,close).

%--- Def. on real domain.
%% Observe that this operator has not into account the second element.
%% (A :&: B) is equivalent to :&:(A)
%% For syntax reasons we maintain the binary charateristics of the
%% operator. A unary sqrt operator is defined below

:&:(Ele1,Ele2,Ele3):-lattice(Ele1,real), lattice(Ele2,real),
                     Ele3 is  sqrt(Ele1).

% ------------------- Unary Sqrt operator -----------------------------
:-dynamic ':$:'/2.
operator(625,fx,':$:').

%--- Mode declaration
declara(:$:,L,L).       %% ':$:' :: L  -> L
```

```
%--- Def. on Bracket domain
:$:(open,open).
:$:(close,close).

%--- Def. on real domain.
:$:(Ele1,Ele3):-lattice(Ele1,real), Ele3 is  sqrt(Ele1).

% --------- Unary Operator to inverse the bracket of a value ----------
:-dynamic ‘:@@@:’/3.
operator(625,fx,‘:@@@:’).

%--- Mode declaration
declara(:@@@:,L,L).       %% ‘:@@@:’ :: L -> L

%--- Def. on Bracket domain
:@@@:(open,close).
:@@@:(close,open).

%--- Def. on any domain except the bracket domain.
:@@@:(P1,P1):-not(lattice(P1,bracket)).
%% Examples
%%         :@@@: 1] = 1)
%%         :@@@: 1) = 1]

% --------- Operator to unary minus ----------------------------------
:-dynamic ‘:--:’/3.
operator(625,fx,‘:--:’).

%--- Mode declaration                          _
declara(:--:,mirror(L),L).       %% ‘:--:’ :: L  -> L

%--- Def. on Bracket domain
:--:(open,open). :--:(close,close).

%--- Def. on numeric domains.
:--:(A,B):-lattice(A,integer),lattice(B,integer),B is -A.
:--:(A,B):-lattice(A,real),lattice(B,real),B is -A.
```

Some operators for combined domains

```
%------------------ COMBINING OPERATORS   ----------------------------
% Operator that combines two elements to form a compound element.
```

```
:-dynamic ‘:@:’/3.
operator(625,xfx,‘:@:’).

%--- Mode declaration
declara(:@:,L1,L1,_L2).       %% ‘:@:’ :: L1 x L1 -> L2

%--- Def. on Bracket domain
:@:(open,_,open).
:@:(close,B,B).

%--- Def. on INTEGER point domain.
:@:(Ele1,Ele2,(Ele1,Ele2)):-lattice(Ele1,integer),
                            lattice(Ele2,integer),!.

%--- Def. on REAL point domain.
:@:(Ele1,Ele2,(Ele1,Ele2)):-lattice(Ele1,real), lattice(Ele2,real),!.

%--- Def. on rectangle domain.
:@:(Ele1,Ele2,(Ele1,Ele2)):-lattice(Ele1,int_point),
                            lattice(Ele2,int_point),!.

% --------- Operator to combine points to form a rectangle-------------
:-dynamic ‘:=:’/3.

operator(625,xfx,‘:=:’).

%--- Mode declaration
declara(:=:,L,L,L).       %% ‘:=:’ :: L x L -> L

%--- Def. on Bracket domain
:=:(open,_,open).
:=:(close,B,B).

%--- Def. on point domain.
:=:(P1,P2,(P1,P2)):-lattice(P1,int_point),
                    lattice(P2,int_point),!.
```

Appendix C

High Level Constraints in $clp(\mathcal{L})$

Some basic constraints

```
%%---------------------- A generic 'equality' constraint ---------------
operator(650,xfx,=:).
constraints (=:)/2.
X =: Y <=> (X in (min Y)..(max Y),
            Y in (min X)..(max X)).

%%---------------------- A generic 'non-equal' constraint --------------
operator(650,xfx,\=:).
constraints (\=:)/2.
X \=: Y <=> (var(X),ground(Y))
        | (X in (bottom,close)..(Y,open);
           X in (Y,open)..(top,close)
          ).
Y \=: X <=> (var(X),ground(Y))
        | (X in (bottom,close)..(Y,open);
           X in (Y,open)..(top,close)
          ).
X \=: Y <=> (ground(X),ground(Y))
        | (X \== Y).

%%--------------- A generic 'less or equal than' constraint-------------
operator(650,xfx,<=:).
constraints (<=:)/2.
X <=: Y <=> (X in (bottom,close)..(max Y),
             Y in (min X)..(top,close) ).

%%--------------- A generic 'greater or equal than' constraint----------
operator(650,xfx,>=:).
```

```
constraints (>=:)/2.
X >=: Y <=> Y <=: X.

%%-------------- A generic 'less than' constraint----------------------
operator(650,xfx,<:).
constraints (<:)/2.
X <: Y <=> X <=: Y, X \=: Y.

%%An alternative more complex (and incomplete) definition
%X <: Y <=> (var(X),var(Y))
%      | X in (bottom,close)..(:@@@:(max Y)),
%        Y in (:@@@:(min X))..(top,close).
%X <: Y <=> (ground(X),var(Y))
%      | Y in (X,open)..(top,close).
%X <: Y <=> (var(X),ground(Y))
%      | X in (bottom,close)..(Y,open).
%X <: Y <=> (ground(X),ground(Y))
%      | X @< Y.

%%-------------- A generic 'greater than' constraint-------------------
operator(650,xfx,>:).
constraints (>:)/2.
X >: Y <=> Y <: X.
```

(Overloaded) arithmetic constraints

```
%%-------------- A 'plus' constraint-----------------------------------
constraints plus/3.
plus(X,Y,Z) <=> (X in ((min Z):-:(max Y))..((max Z):-:(min Y)),
                 Y in ((min Z):-:(max X))..((max Z):-:(min X)),
                 Z in ((min X):+:(min Y))..((max X):+:(max Y))
                ).

%%-------------- A 'substraction' constraint--------------------------
constraints subs/3.
subs(X,Y,Z) <=> plus(Y,Z,X).

%%-------------- A 'times' constraint---------------------------------
%% Note that the operator ':/:' controls the case of division by zero
constraints times/3.
times(X,Y,Z) <=> (X in ((min Z):/:(max Y))..((max Z):/:(min Y)),
                  Y in ((min Z):/:(max X))..((max Z):/:(min X)),
                  Z in ((min X):*:(min Y))..((max X):*:(max Y))
```

```
                ).

%%-------------- A 'divide' constraint---------------------------------
constraints divide/3.
divide(X,Y,Z) <=> times(Y,Z,X).

%%-------------- A 'negation' constraint-------------------------------
%%Next constraint is true if X=-Y on numeric domains.
constraints neg/2.
neg(X,Y) <=> (X in (:--:(max Y))..(:--:(min Y)),
              Y in (:--:(max X))..(:--:(min X))).
```

Boolean solvers

Boolean solvers can be coded by following the schema shown in (Codognet and Diaz, 1996a). For example, here we show the constraint *not*/2.

```
not(X,Y)  <=> [X,Y] in (0,close)..(1,close),
               X in ((1,close):-:(val Y))..((1,close):-:(val Y)),
               Y in ((1,close):-:(val X))..((1,close):-:(val X)).
```

Symbolic constraints

```
constraints and0/2.
and0(X,Y) <=> and(X,Y,0).              %% 0 = X and Y

constraints and0/3.
and0(X,Y,Z) <=> and(X,Y,W),and0(W,Z).  %% 0 = X and Y and Z

constraints or1/2.
or1(X,Y) <=> or(X,Y,1).                %% X or Y = 1

at_least_one(L):-at_least_one1(L,1).

at_least_one1([X],X).
at_least_one1([X|L],R):-at_least_one1(L,R1), or(X,R1,R).

at_most_one([]).
at_most_one([X|L]):-not_two(L,X), at_most_one(L).

not_two([],_).
not_two([X1|L],X):-and0(X1,X), not_two(L,X).
```

```
only_one(L):-at_least_one(L), at_most_one(L).
```

Appendix D

$clp(\mathcal{L})$ Programs

Computing the e number

The following program computes the e number. Observe the use of the high level constraints 'divide'/3, 'plus'/3 and '=:'/2.

```
e(N,E):- Err is exp(10,-(N+2)),
         Half is 1/2,
         inv_e_series(Half,Half,3,Err,Inv_E),
         divide(1.0,Inv_E,E).

inv_e_series(Term,S0,_,Err,Sum):-abs(Term) =< Err,!,
                                 S0 =: Sum.

inv_e_series(Term,S0,N,Err,Sum):-N1 is N+1,
           Term1 is -Term/N,
           plus(Term1,S0,S1),
           inv_e_series(Term1,S1,N1,Err,Sum).
```

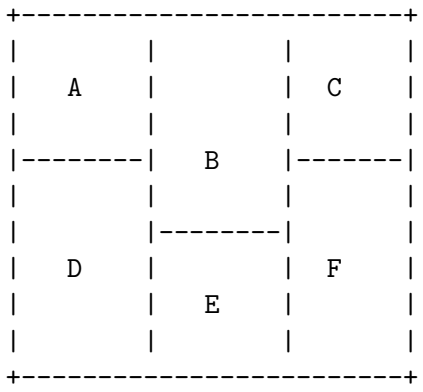
An example of resolution in the $clp(\mathcal{L})$ command line is shown below:

```
clp(L) > e(100,N).
    yes.
    N = 2.718281828459044
 ? |:
```

Graph colouring

Colour the map shown below with three colors in such a way that no two adjacent regions are colored with the same color. Observe the use of the high level constraint '\=:'/2, and also note that constraint solving is executed directly over the colour

domain, instead of the usual solution of defining a mapping from the colour to the integer domain and solving the problem in the integer domain.




```
colour([A,B,C,D,E,F]):-
      domain([A,B,C,D,E,F],white,orange),
      A\=:B,  A\=:D,  B\=:C,
      B\=:D,  B\=:E,  B\=:F,
      C\=:F,  D\=:E,  E\=:F,
      labeling([A,B,C,D,E,F]).
```

An example of resolution in the clp($\mathcal{L}$) command line is shown below:

```
clp(L) > colour(L).
         yes
         L = [white,yellow,white,orange,white,orange] ;
         yes
         L = [white,orange,white,yellow,white,yellow]
         ...
```

Newton's approximations to solve square roots

The following benchmark implements N steps of Newton's approximation for the square root function at point 2. Observe the use of the high level constraints '=:', 'divide/3' and 'plus/3'.

```
root(N,R):-root(N,1.0,R).

root(0,S,R):-!, S=:R.
root(N,S,R):-N1 is N-1,
             divide(S,2.0,S2aux),
             divide(1.0,S,S1aux),
             plus(S2aux,S1aux,S1),
             root(N1,S1,R).
```

Here we show some examples of resolution on the $clp(\mathcal{L})$ command line.

```
clp(L) > root(1,R).
    yes.
    R = 1.5
 ? |:
   no
clp(L) > root(2,R).
    yes.
    R = 1.4166666666666665
 ? |:
   no
clp(L) > root(3,R).
    yes.
    R = 1.4142156862745097
clp(L) > root(4,R).
    yes.
    R = 1.4142135623746899
```