Spacetime Programming: A Synchronous Language for Constraint Search

Pierre Talbot

▶ **To cite this version:**

Pierre Talbot. Spacetime Programming: A Synchronous Language for Constraint Search. Programming Languages [cs.PL]. Sorbonne Université, Université Pierre et Marie Curie, Paris 6, 2018. English. NNT : . tel-02924854v1

HAL Id: tel-02924854

https://hal.science/tel-02924854v1

Submitted on 11 Jul 2018 (v1), last revised 28 Aug 2020 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT DE SORBONNE UNIVERSITÉ

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Pierre Talbot

Pour obtenir le grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Sujet de la thèse :

Spacetime Programming

A Synchronous Language for Constraint Search

dirigée par Carlos Agon et encadrée par Philippe Esling

soutenue le 6 juin 2018 devant le jury composé de :

M. Carlos Agon	Directeur de thèse
Mme. Emmanuelle Encrenaz	Examinatrice
M. Antoine Miné	Examinateur
M. Vijay A. Saraswat	Rapporteur
M. Manuel Serrano	Examinateur
M. Sylvain Soliman	Examinateur
M. Guido Tack	Rapporteur
M. Peter Van Roy	Examinateur

Spacetime Programming

A Synchronous Language for Constraint Search

Pierre Talbot

2018

Abstract

Constraint programming is a paradigm for computing with mathematical relations named constraints. It is a declarative approach to describe many real-world problems including scheduling, vehicles routing, biology and musical composition. Constraint programming must be contrasted with procedural approaches that describe *how* a problem is solved, whereas constraint models describe *what* the problem is. The part of *how* a constraint problem is solved is left to a general constraint solver. Unfortunately, there is no solving algorithm efficient enough to every problem, because the search strategy must often be customized *per problem* to attain reasonable efficiency. This is a daunting task that requires expertise and good understanding on the solver's intrinsics. Moreover, higher-level constraint-based languages provide limited support to specify search strategies.

In this dissertation, we tackle this challenge by designing a programming language for specifying search strategies. The dissertation is constructed around two axes: (i) a novel theory of constraint programming based on lattice theory, and (ii) a programming language, called *spacetime programming*, building on lattice theory for its data structures and on synchronous programming for its computational model.

The first part formalizes the components of inference and search in a constraint solver. This allows us to scrutinize various constraint-based languages through the same underlying foundations. In this respect, we give the semantics of several paradigms including constraint logic programming and concurrent constraint programming using lattices. The second part is dedicated to building a practical language where search strategies can be easily composed. Compositionality consists of taking several distinct strategies and, via some operators, to compose these in order to obtain a new strategy. Existing proposals include extensions to logic programming, monadic constraint programming and search combinators, but all suffer from different drawbacks as explained in the dissertation. The original aspect of spacetime programming is to make use of a temporal dimension, offered by the synchronous paradigm, to compose and synchronize search strategies on a shared logical clock.

This paradigm opens the door to new and more complex search strategies in constraint programming but also in applications requiring backtracking search. We demonstrated its usefulness in an interactive computer-aided composition system where we designed a search strategy to help the composer navigating in the state space generated by a musical constraint problem. We also explored a model checking algorithm where a model dynamically generates a constraint satisfaction problem (CSP) representing the reachability of some states. Although these applications are experimental, they hint at the suitability of spacetime programming in a larger context.

Acknowledgements

Je remercie très chaleureusement Emmanuel Chailloux qui, après un Skype mémorable, m'a accepté pour ma dernière année en Master, et qui m'a trouvé un stage qui allait me diriger vers une thèse. Plutôt que de parler de stage, c'est plutôt vers Carlos, ou Carlito, après 4 ans on se le permet, et Philou, que j'ai été dirigé. Une histoire qui a commencé à "la salle de réunion", et dans laquelle ils m'ont prodigué des conseils des plus avisés, et ce toujours dans une excellente ambiance.

Merci aussi à tous mes bro'ffices. Clément et Mattia avec qui tout ça a commencé. Merci aux amis de Jussieu, et surtout l'équipe APR, Steven, Pierrick, Vincent, qui étaient là aux premières heures. Merci Rémy, tu m'as suivi sur les toutes les branches de la A, et puis même jusqu'à Châtelet (c'est ça la force de l'amitié). L'amicale de ping pong de l'IRCAM : merci à Gérard, Sylvie, Grégoire, Karim, Cyrielle, Florence, Olivier et tous les pongistes du −4. Et puis, merci Jean-Louis, Pierre N°2, et tous les autres.

I would like to acknowledge Vijay Saraswat, Guido Tack and Antoine Miné for their valuable comments and corrections on my dissertation.

I had the opportunity to go to Japan twice, and I am grateful for all the help Philippe Codognet provided me. Chiba-sensei and Phong Nguyen, warmly welcomed me to their labs in Japan, and I highly appreciate their support. Of course, special thanks to all of my friends in Japan: Marceau Suntory, Klervie, Nate, Mathieu Green Trompette, Nicolas and Guillaume. I would like to thank Thomas Silverston too, for the good time we spent together. Thanks to all these people, I had a very pleasant and inspiring stay in Tokyo.

I want to thank my wife, Yinghan, who continuously supports me, and always has faith in me.

Merci aussi à toute ma famille, ma mère, mon père, Céline et Thomas, pour m'avoir encouragé, et ce même parfois de l'autre bout du monde.

Pierre Talbot
Paris, le 17 avril 2018

Contents

Introduction **1**

I A Lattice Theory of Constraint Programming **9**

1 Lattice Hierarchy for Constraint Solving **10**
1.1 Constraint programming . . . 11
1.1.1 Constraint modelling . . . 11
1.1.2 Constraint solving . . . 13
1.1.3 Propagation . . . 16
1.2 Lattice structure . . . 18
1.3 Lattice hierarchy for constraint programming . . . 27
1.3.1 L_0: Value . . . 28
1.3.2 L_1: Domain . . . 28
1.3.3 L_2: Variable store . . . 29
1.3.4 L_3: Constraint satisfaction problem . . . 29
1.3.5 L_4: Search tree . . . 32
1.3.6 L_5: Tree store . . . 33
1.3.7 L_n: Algorithm selection . . . 35
1.4 Inference in L_3 with propagation . . . 35
1.4.1 Modelling a problem . . . 35
1.4.2 Deciding . . . 36
1.4.3 Propagating . . . 37
1.5 Bridging L_3 and L_4 with backtracking . . . 38
1.5.1 Delta operator . . . 40
1.5.2 Queueing strategy . . . 41
1.5.3 Branching strategy . . . 43
1.5.4 Exploration strategy . . . 45
1.6 Inference in L_4 with constraint optimization problem . . . 46
1.7 Pruning in L_4 with backjumping . . . 47
1.8 Conclusion and discussion . . . 50

2 Constraint-based Programming Languages **52**
2.1 Introduction . . . 52
2.2 Background . . . 54
2.2.1 First-order logic . . . 55
2.2.2 Semi-decidability . . . 56
2.2.3 Logic programming . . . 57
2.2.4 Decidable fragments of FOL . . . 59
2.3 Modelling languages: from L_0 to L_4 ($\sqcup$) . . . 60
2.3.1 Declarative languages . . . 60
2.3.2 Constraint imperative programming . . . 65
2.4 Inference-based languages: from L_0 to L_3 ($\sqcup, \models$) . . . 67
2.4.1 Concurrent logic programming . . . 67
2.4.2 Concurrent constraint programming . . . 70
2.5 Bridging L_3 and L_4 ($\sqcup, \models, \Delta$) . . . 78
2.5.1 Andorra-based languages . . . 78
2.5.2 Constraint logic programming . . . 81
2.5.3 Lattice-based semantics of CLP . . . 86
2.6 Search languages: L_4 and above ($\sqcup, \models, \Delta$) . . . 91
2.6.1 Search in logic languages . . . 92
2.6.2 Search in other paradigms . . . 97
2.6.3 Control operators for search . . . 101
2.7 Conclusion and discussion . . . 103

II Spacetime Programming **106**

3 Overview of Spacetime Programming **107**
3.1 Synchronous programming . . . 107
3.1.1 Linear time . . . 108
3.1.2 Causality analysis . . . 109
3.2 Concepts of spacetime programming . . . 111
3.2.1 Branching time . . . 111
3.2.2 Blending linear and branching time . . . 112
3.2.3 Host variables as lattice structures . . . 113
3.3 Syntax and intuitive semantics . . . 114
3.3.1 Communication fragment . . . 114
3.3.2 Undefinedness in three-valued logic . . . 117
3.3.3 Synchronous fragment . . . 118
3.3.4 Search tree fragment . . . 119
3.3.5 Read-write scheduling . . . 120
3.3.6 Derived statements . . . 121
3.4 Programming search strategies in L_4 . . . 122
3.4.1 A minimal constraint search algorithm . . . 122

3.4.2 Branch and bound 125
3.5 Hierarchy in spacetime with universes 126
3.5.1 Universe fragment 127
3.5.2 Communication across layers of the hierarchy 128
3.6 Programming in L_5 with universes 130
3.6.1 Iterative deepening search 130
3.6.2 Branch and bound revisited 131

4 Behavioral Semantics Inside an Instant 134
4.1 Behavioral semantics 134
4.2 Composition of search trees 137
4.3 Expressions and interface with host 139
4.4 Statements rules 141
4.4.1 Communication fragment 141
4.4.2 Synchronous fragment 143
4.4.3 Search tree fragment 145
4.5 Causality analysis 146
4.5.1 Logical correctness 146
4.5.2 A constraint model of causality 147
4.5.3 From the spacetime program to the causality model 150
4.5.4 Causal dependencies 150
4.5.5 Causal statements 152
4.6 Conclusion and discussion 156

5 Behavioral Semantics Across Instants 159
5.1 Composition of parallel universes 160
5.2 Universe hierarchy 160
5.3 Hierarchical behavioral semantics 163
5.4 Hierarchical variable 164
5.4.1 Read, write and allocate variables in the hierarchy 165
5.4.2 Top-down spacetime transfer 167
5.4.3 Bottom-up spacetime transfer 169
5.5 Semantics of the rules across time 169
5.5.1 Semantics of the queueing strategy 169
5.5.2 Universe statement 171
5.5.3 Reaction rule 172

III Applications 173

6 Modular Search Strategies 174
6.1 Pruning strategies 174
6.1.1 Statistics 174
6.1.2 Bounded search 175

6.2 Restart-based search strategies . . . 177
6.2.1 L_5 search strategies . . . 177
6.2.2 Exhaustiveness and state space decomposition . . . 178
6.2.3 Composing L_5 search strategies . . . 181
6.2.4 L_6 search strategy . . . 183
6.3 Guarded commands . . . 184

7 Interactive Computer-Aided Composition 186
7.1 Introduction . . . 186
7.2 Score editor with constraints . . . 187
7.2.1 Visual constraint solving . . . 187
7.2.2 Spacetime for composition . . . 188
7.3 Interactive search strategies . . . 189
7.3.1 Stop and resume the search . . . 189
7.3.2 Lazy search tree . . . 190
7.3.3 Diagnostic of musical rules . . . 191
7.4 Conclusion . . . 192

8 Model Checking with Constraints 194
8.1 Model checking . . . 194
8.2 Model checking with constraints . . . 197
8.2.1 Constrained transition system . . . 197
8.2.2 Existential constraint system . . . 199
8.2.3 Lattice abstraction . . . 200
8.3 Spacetime algorithm . . . 202
8.4 Search strategies for verification . . . 204
8.4.1 Verifying constraint-based property . . . 204
8.4.2 Constraint entailment algorithm . . . 205
8.5 Conclusion . . . 207

IV Conclusion 208

9 Conclusion 209

10 Future work 211
10.1 Towards constraint programming with lattices . . . 211
10.2 Extensions of spacetime . . . 212
10.3 Beyond search in constraint programming . . . 215

Bibliography 216

Introduction

This dissertation links the paradigms of constraint programming and synchronous programming. We present these two fields in turn, and then we outline spacetime programming, the language proposed in this work for unifying these two paradigms.

Constraint programming

In theoretical computer science, a function models a correspondence between the sets of input and output states, while a relation expresses a property that must be satisfied by the computation. Operationally, functions and relations are embodied in deterministic and non-deterministic computations. From an expressive standpoint, this distinction is artificial since any algorithm described by a non-deterministic algorithm can be described by a deterministic equivalent one.[1] So what is the advantage of using one model of computation over the other? The answer is that sometimes it is much easier to express *what* properties a solution must satisfy than to specify *how* we should compute this solution. This dual view is at the heart of the differences between imperative and declarative programming paradigms, which allow us to program respectively the *how* and the *what*. Around 1970, logic programming emerged as a paradigm conciliating these two views, where a declarative program also has an executable semantics [Kow74]. In this dissertation, we focus on a generalization of the logic computational model called *constraint programming*.

Constraint programming is a paradigm for expressing problems in terms of mathematical relations—called constraints—over variables (e.g. $x > y$). Constraints are an intuitive approach to naturally describe many real-world problems which initially emerged as a subfield of artificial intelligence and operational research. The flagship applications in constraint programming encompass scheduling, configuration and vehicles routing problems; all of these three have a chapter in [RvBW06]. Constraints are also applied to various other domains such as in music [TA11, TAE17], biology [RvBW06] and model checking [CRVH08, TP17]. Besides its application in practical fields, constraint programming is also the theory of efficient algorithms for solving constraint satisfaction problems (CSP).

[1] However, the complexity of these two algorithms might not be equivalent; answering this question is the famous P$\neq$NP problem.

In this dissertation we are interested in *how* a CSP is solved. Constraint solving usually relies on a backtracking algorithm enumerating the values of the variables until a solution is found, i.e. all the constraints are satisfied. This algorithm generates a *search tree* where every branch reflects a choice made on one variable. In case of unsatisfiability, the algorithm backtracks in the search tree to the previous choice and try another branch. For example, consider the CSP with two variables x and y both taking a value in the set $\{1, 2\}$ (called the *domain* of the variable), and the constraint $x \neq y$. We first assign x to 1, and then y to 1 which result in an unsatisfiable CSP because $x \neq y$ does not hold. Therefore, we backtrack in the search tree by trying another branch which assigns y to 2. This time we obtain a solution where $x = 1$ and $y = 2$. The algorithm exploring the search tree is called a *search strategy*.

Synchronous programming

The synchronous paradigm [Hal92] proposes a notion of logical time dividing the execution of a program into a sequence of discrete instants. The main goal of logical time is to coordinate processes that are executed concurrently while avoiding typical issues of parallelism, such as deadlock or indeterminism [Lee06]. During one instant, processes execute a statically bounded number of actions—unbounded loop or recursion are prohibited—and wait for each other before the next instant.

Although the programmer views a synchronous program as a set of processes evolving more or less independently, the processes are scheduled sequentially at runtime. In other words, every instruction of each process happens before or after another one. An important aspect of the synchronous model is that a causality analysis ensures programs to be deterministic (resp. reactive): at most (resp. at least) one output is produced from one input. It implies that even if the program can be sequentialized in different ways, it will always produce a unique result.

Concretely, we can implement a synchronous program as a coroutine: a function that maintains its state between two calls. A call to the coroutine triggers the execution of one instant. We must precise that the coroutine is generally called from a host language interfacing between the user environment and the synchronous program.

In this dissertation, we focus on the imperative synchronous language Esterel [Ber00b]. The input and output data of an Esterel program are boolean variables called signals. A simple example of an Esterel program is a watch: its state changes when the user presses buttons or when a second has elapsed. We can imagine the buttons of the watch being boolean variables, and every process implements the logic of one button. The causality analysis warns the programmer about corner cases that he forgot to implement. For example, if the user presses two buttons at the same time, and that their processes produce two output states that are conflictual.

Motivation

On the language side of constraint programming, pioneered by logic programming, an influential paper of Kowalski captures the *how* and the *what* in the equation "Algorithm = Logic + Control" [Kow79]. In the early day of logic programming, a hope was to keep the control aspect hidden from the programmer who would only specify the logic part of its program. Forty years have passed by but a solving algorithm that is efficient for every problem is yet to be found. We quote Ehud Shapiro [Sha86] who already commented on this illusory hope back in 1986:

> The search for the ultimate control facility is just the search for the "philosopher's" stone in disguise. It is similar to the search for the general problem-solver, or for the general efficient theorem-prover, which simply does not exist. We have strong evidence today that many classes of seemingly simple problems have no general efficient solution.

The logic part has been extensively studied with constraint modelling languages [Lau78, VHMPR99, NSB+07] and it is well-understood today. However, while these languages are outstanding for the logic specification of combinatorial problems (often NP-complete), the specification alone generally leads to poor performances. This is why some limited form of controls exist in Prolog, such as the controversial *cut* operator that prunes the search tree. Numerous approaches have been designed to provide language abstractions for specifying the control part of constraint solving [VH89, LC98, VHML05]. A fundamental question in these approaches is the compositionality of search strategies: how can we easily combine two strategies and form a third? Recently, we witness a growing number of proposals [SSW09, STW+13, MFS15] that attempt to cope with this compositionality issue. This problematic is central in the design of the spacetime programming language developed over this dissertation.

A search language is important because experience has proven that reducing the solving time of combinatorial problems, especially in industry, must be tackled per problem or even per problem-instance after the design and test of several solver configurations and search strategies [SO08, TFF12]. Therefore, we need to easily try and test the efficiency of new search strategies for a given problem. Currently, the implementers of search strategies have to choose between limited but high-level specification languages and full-fledged but low-level constraint libraries. The first family encompasses constraint logic programming where control is obtained with predetermined building blocks assembled within a search predicate. For example, GNU-Prolog [DAC12] proposes the predicate `fd_labelling/2` where the first argument is the list of variables and the second some options for selecting the variables and values enumeration strategies. The second family concerns constraint libraries such as Choco [PFL15] or GeCode [STL14] which are designed to be extensible and opened to search programming while remaining very efficient. The main problem of the library approach is that users need good understanding of the intrinsics of

the library but only experts are knowledgeable in this domain. Of course, many works attempt to bridge the gap between these two extreme categories, which we will discuss in Chapter 2.

Overview

In Part I, we contribute to a theoretical framework of CSPs based on lattice theory. Part II formalizes the syntax and semantics of spacetime programming based on the lattice framework developed in the first part. Finally, in Part III, we develop three applications of search strategies written in the spacetime paradigm. These applications develop on the interactive and hierarchical aspects of spacetime programming. We outline the content of each part in the following paragraphs.

Part I: A lattice theory of constraint programming

In Chapter 1, we formulate a lattice theory of constraint programming for *finite domains* where constraint inference is based on the denotational model introduced in [Tac09]. As far as we know, our theory is the first attempt to formalize the structures underlying the inference and search components in a unified framework. The main idea is to define a hierarchy of lattices where each level is built on the powerset of the previous level. This hierarchical structure is central to constraint programming, as the notions of domain, constraint and search tree depend on each other. In Chapter 2, we review a large spectrum of constraint-based programming languages. A new aspect of this study is to classify the various languages within the hierarchy levels developed in the former chapter. To demonstrate the generality of our lattice hierarchy, we develop the lattice-based semantics of concurrent constraint programming [Sar93] and constraint logic programming [MJMS98]. We pinpoint several issues in modern search languages that are tackled by spacetime programming later in our proposal.

Part II: Spacetime programming

In order to tackle the compositionality issue, our approach is to link constraint programming and synchronous programming in a unified language, called *spacetime programming* or "spacetime" for short. We briefly explain its model of computation and how its constructions are relevant to the lattice hierarchy developed in Chapter 1.

Computational model Spacetime inherits most of the temporal statements of Esterel, including the delay, parallel, loop, suspension and abortion statements. We connect the search tree induced by constraint programming and the linear time of synchronous programming with a principle summarized as follows:

A node of the search tree is explored in exactly one logical instant.

Our key observation is that search strategies are synchronous processes exploring a search tree at a same pace. A challenge is introduced by the notion of linear time exploring a tree-shaped space: how does the variables evolve in time and space in a coherent way? We propose to annotate variables with a *spacetime attribute* to distinguish between variables local to one instant (`single_time`), variables global to the whole computation (`single_space`) and backtrackable variables local to a branch in the search tree (`world_line`). For example, a global variable can be used to count the number of nodes in the search tree, and a backtrackable variable can be the constraint problem itself.

Variables defined on lattices The second characteristic of spacetime is inherited from concurrent constraint programming (CCP) [Sar93] where the memory is defined by a global constraint store accumulating partial information. Processes can communicate and synchronize through this store with two primitives: $tell(c)$ for adding a constraint c into the store and $ask(c)$ for asking if c can be deduced from the store. Concurrency is treated by requiring the store to grow *monotonically*, i.e. removal of information is not permitted. The difference of spacetime with CCP is that variables are all defined over lattice structures, instead of having a central and global constraint store. The tell and ask operations in spacetime are defined on lattices, where tell relies on the least upper bound operation and ask relies on the order of the lattice. These lattice operations effectively relate the framework developed in Chapter 1 and the spacetime language. Besides, we must note that the lattice structures are programmed in the host language of spacetime. This allows the user of spacetime to reuse existing constraint solvers which are abstracted as lattice variables.

Causality analysis From a synchronous perspective, every instant of a spacetime process is a monotonic function over its lattice variables. This property is ensured by the causality analysis of spacetime (Section 4.5). It is an important contribution because current synchronous languages are defined on a restricted class of lattices, called *flat lattices*. We refine the behavioral semantics of Esterel [Ber02] to lattice-based variables which imply scheduling read and write operations correctly. A correct scheduling is crucial to preserve the deterministic and reactive properties of a synchronous program.

Hierarchical computation The fourth concept synthesizes time hierarchy from synchronous programming and spatial hierarchy from logic programming. Time hierarchies were first developed in Quartz [GBS13] and ReactiveML [Pas13, MPP15] to execute a process on a faster time scale. They propose that in one instant of the global clock, we can execute more than one local step of a process. Spatial hierarchies were introduced with logic programming and more particularly with deep guards and computation spaces in the Oz programming language [Sch02]. It executes a process locally with its own queue of nodes.

We propose the concept of *universe* which encapsulates the spatial and time dimensions of a computation. Basically a universe extends a computation space to a clock controlling the evolution of its variables through time. In the lattice hierarchy, a universe enables users to create an additional layer in the hierarchy. This extension is particularly useful to write search strategies that restart the search or explore several times a search tree. For example, by encapsulating a process in a universe exploring a search tree, we can manipulate a collection of search trees, where in each instant we explore a new search tree. The concept of universe essentially implements the powerset derivation defining the levels of the lattice hierarchy.

Part III: Applications

This last part studies three applications of spacetime programming built on search strategies developed in the spacetime paradigm. In Chapter 6, we show how to design a modular library of search strategies. It also illustrates that spacetime is applicable to specialized and complex search strategies in the field of constraint programming. Beyond constraint programming, spacetime is also suited to implement interactive applications. In particular, we introduce in Chapter 7 an interactive search strategy for a computer-aided composition system allowing the composer to dynamically navigate in the state space generated by a musical CSP. It attempts to put back constraints at the heart of the compositional process instead of solving musical problems in a black box fashion. Finally, we expand the scope of spacetime programming to model checking where we explore a new algorithm based on a constraint store to solve parametric model checking problems (Chapter 8).

Summary

To summarize, we introduce a programming language, called *spacetime programming*, to control the evolution of lattice structures through a notion of logical time. To demonstrate the capabilities of our proposal, we study CSPs as lattice structures, which is a general framework for specifying and solving combinatorial problems. A long-standing issue in this field, started with logic programming in the seventies, is how to compose two existing search strategies. We demonstrate that logical time is an excellent abstraction to program and combine search strategies in order to solve a combinatorial problem. We extend the scope of this paradigm by investigating an interactive computer-aided composition software and a model checking algorithm based on constraints.

Contributions

- A hierarchical lattice theory **unifying the modelling and search** components (i.e. *what* and *how*) of constraint programming (Chapter 1).
- A **survey of constraint-based programming languages** that classifies and formalizes constraint languages according to the lattice hierarchy (Chapter 2). The main insight is that many constraint-based languages can be defined over the same structure, which helps in better understanding their differences.
- The semantics of spacetime programming provides the foundations for **composing search strategies** which are synchronous processes over the lattice hierarchy (Chapters 3, 4 and 5). The originality is to map logical time to the exploration of the search tree.
- We propose a **causality analysis of a spacetime program** for variables defined over lattices. It extends the causality analysis of Esterel which is defined for boolean variables only.
- We show that **modular, interactive and hierarchical search strategies** can be programmed in the spacetime language (Chapters 6, 7, and 8).

How to read this document

This dissertation is organized in 10 chapters that can be read in different ways. We show the dependencies between the different chapters or parts in Figure 1. For a short tour of the dissertation, we suggest to read Section 1.3 for the lattice foundation, Chapter 3 for the overview of spacetime, and Chapter 7 for a more in-depth example of an interactive search strategy for musical composition. The survey in Chapter 2 can be read apart from the spacetime language.

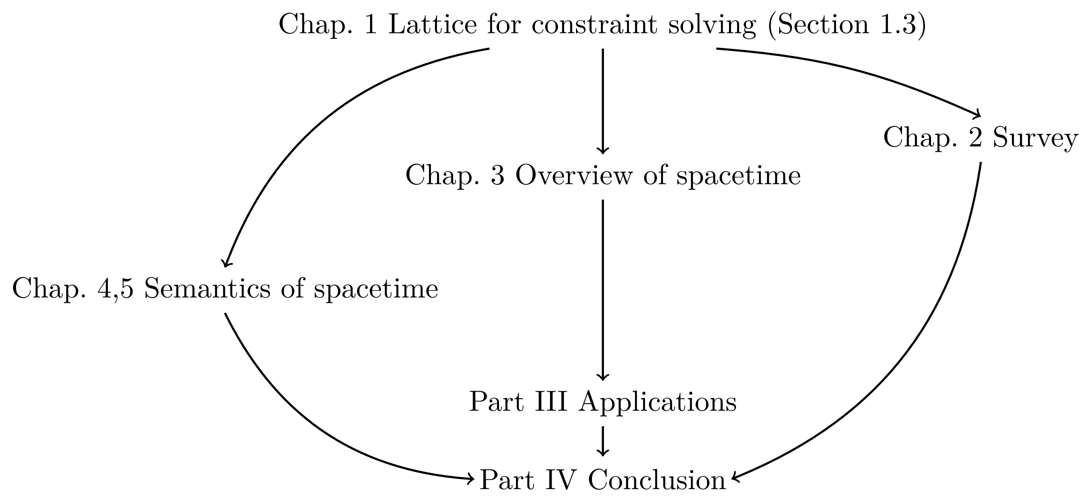


Figure 1: Organization of the dissertation.

Part I

A Lattice Theory of Constraint Programming

Chapter 1

Lattice Hierarchy for Constraint Solving

Over the years, numerous techniques have been devised to solve constraint problems and to improve the efficiency of existing algorithms. They range from low-level considerations such as memory management to higher-level techniques, such as propagating constraint, learning new constraints, and intelligently splitting a state space. The current foundation of constraint programming is built on a set theoretic definition of the constraint problem, which is explained in Section 1.1. However, solving techniques frequently use more complex combinatorial objects rather than the constraint problem structure alone. A prime example of such objects is the search tree—a structure to enumerate all the solutions, which is implicitly generated by the solving algorithm only. Since every algorithm is different, we do not have a common ground to understand the subtle differences among these techniques in terms of how they cooperate and integrate with each other. Therefore, in order to understand the relations among available techniques, we believe that a more extensive foundation of constraint programming is needed.

The purpose of this chapter is to devise a new foundation for constraint programming, in which all the combinatorial structures are explicitly defined. To achieve this, we rely on the theory of lattices to precisely capture the combinatorial objects underlying constraint solving. To make this chapter self-contained, we introduce the basics of lattice theory in Section 1.2. Next, we organise the combinatorial objects of constraint programming in a hierarchy where each level relies on the levels before it (Section 1.3). For example, relations are defined onto variables; a search tree is a collection of partially solved constraint problems. Once this structure defined, we present several key constraint solving techniques as monotonic functions over this hierarchy. Furthermore, with an algebraic view of lattices, we demonstrate that most algorithms are defined by a combination of three lattice operators. This finding provides a common ground for many techniques including constraint propagation (Section 1.4), backtracking search (Section 1.5), optimization problems (Section 1.6) and backjumping (Section 1.7). Of course, as discussed in Section 1.8, there are more techniques that are interesting to study through the lattice theory. We limit the scope of this chapter to exhaustive solving algorithms

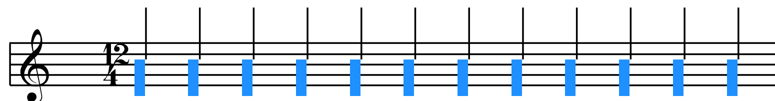
over finite domains, and thus finite lattices, which is one of the most successful settings of constraint programming.

1.1 Constraint programming

Constraint programming is a declarative paradigm for solving constraint satisfaction problems (CSPs) [RvBW06]. The programmers only need to declare the structure of a problem with constraints (e.g. $x > y \wedge x \neq z$) and it is automatically solved for them. We illustrate the modelling aspect of constraint programming in Section 1.1.1 with a musical problem. Under the hood, constraint solving usually relies on a backtracking algorithm enumerating the values of the variables until we find a solution, i.e. all the constraints are satisfied. We introduce a mathematical framework for solving finite CSPs in Section 1.1.2 which is based on the dissertation of Guido Tack [Tac09]. In contrast, other heuristic methods such as local search, not guaranteeing to find a solution, are left out of scope in this dissertation. What's more, in Section 1.1.3, we focus on a propagation-based algorithm which is a technique used in almost every constraint solver (e.g. [STL14, PFL15, GJM06, IBM15]). The lattice framework developed in the rest of this chapter is built on the definitions introduced here.

1.1.1 Constraint modelling

As a first example of constraint modelling problem, we consider the all interval-series (AIS) musical problem[1], for example described in [MS74]. It constrains the pitches of the notes as well as the intervals between two successive pitches to be all different. Initially, the pitches are initialized in the interval domain [1..12]. The domains are pictured by the rectangles on the following score:



Throughout the solving process, the domains become smaller, and are eventually instantiated to a note when we reach a solution. An example of such a solution is:

For the composer, it forms a raw material that can be transformed again using others tools such as OpenMusic [AADR98].

The AIS problem can be specified using a constraint modelling language. We illustrate this using MiniZinc [NSB⁺07] which is one of the most popular modelling languages.

[1] AIS is a classic CSP occurring as the problem number 007 in the library CSPLib [GW99].

```
int: n = 12;
array[1..n] of var 1..n: pitches;
array[1..n-1] of var 1..n-1: intervals;
constraint forall(i in 1..n-1)
                 (intervals[i] = abs(pitches[i+1] - pitches[i]));
constraint forall(i,j in 1..n where i != j)
                 (pitches[i] != pitches[j]);
constraint forall(i,j in 1..n-1 where i != j)
                 (intervals[i] != intervals[j]);
solve satisfy;
```

This code is purely declarative. The arrays `pitches` and `intervals` are respectively storing the pitches and the differences between these pitches. The macro `forall(`r`)(`c`)` generates—at compile time—a conjunction of the constraints c_i for each i in the range expression r. The first `forall` is used to initialize the array `intervals` with the absolute difference between two pitches. Due to the relational semantics of constraints, the equality predicate $=$ ensures that the value of an interval is synchronized with every pair of successive pitches. Hence, we can read the equality in both directions: the pitches constrain the interval and the interval also constrains the pitches. The last two `forall` statements generate inequality constraints ensuring that all pitches and intervals are pairwise distinct. It remains the statement `solve satisfy` which is a "solve button" for obtaining a solution to this model from the system.

One of the principal reason behind the efficiency of a constraint solver is the presence of global constraints. A global constraint is a n-ary relation capturing a structure that abstracts a common modelling sub-problem. For example, the global constraint `alldifferent(`$\{x_1, \ldots, x_n\}$`)` enforces that the set of variables $\{x_1, \ldots, x_n\}$ contains only different values. A catalogue of the most common global constraints has been established [BCR11] and is updated continuously to incorporate the newest constraints. A large body of work in the constraint programming community is the design of efficient algorithms for existing and novel global constraints but it is out of scope in this work. Back to the AIS example, we observe that the last two `forall` generators are actually both enforcing the `alldifferent` global constraints on different arrays. In MiniZinc, we can improve the model with the `alldifferent` global constraints provided by the system:

```
(...)
constraint alldifferent(pitches);
constraint alldifferent(intervals);
solve satisfy;
```

Actually, we initially provided a decomposition of the `alldifferent` constraint into a conjunction of more elementary constraints. It means that any solver supporting these elementary constraints also supports this global constraint. However, in practice, global constraints are usually mapped to more efficient but lower-level algorithms.

We only give a shallow introduction to the art of modelling with constraints.

The interested reader can find more information and references about constraint modelling in [RvBW06].

1.1.2 Constraint solving

In this thesis, we will be mostly concerned with the constraint solving algorithms behind the "solve button". We start by giving the mathematical foundations for constraint solving and several examples. These definitions are based on [Tac09].

Definition 1.1 (Assignment function)**.** *Given a finite set of variables X and a finite set of values V, an assignment function $a : X \to V$ maps every variable to a value. We note the set of all assignments $Asn \stackrel{\text{def}}{=} V^X$ where V^X is the set of all functions from X to V.*

In addition, we define a *partial assignment* which is an assignment such that at least one variable is mapped to more than one value.

Given a set S, we use $\mathcal{P}(S)$ to denote the powerset of S.

Definition 1.2 (Constraint satisfaction problem (CSP))**.** *A CSP is a tuple $\langle d, C\rangle$ where*

- *The domain function $d : X \to \mathcal{P}(V)$ maps every variable to a set of values called the domain of the variable.*
- *The constraint set C is a subset of the powerset $\mathcal{P}(Asn)$.*

An alternative but equivalent presentation of a CSP is the tuple $\langle X, D, C\rangle$ where the set of variables X and of domains D are made distinct while, in our case, it is embedded in the domain function d. In our framework, a constraint $c \in C$ is defined in extension since c is the set of all assignments satisfied by this constraint. Thus, any assignment $a \in c$ is a solution of the constraint c. For example, the constraint $x < y$ is induced by the set $\{a \in Asn \mid a(x) < a(y)\}$.

The domain function itself can be viewed as a constraint $con(d)$ defined by its set of assignments, and an assignment $a \in Asn$ as a domain function:

$$con(d) \stackrel{\text{def}}{=} \{a \in Asn \mid \forall x \in X,\ a(x) \in d(x)\}$$
$$dom(a) \stackrel{\text{def}}{=} \forall x \in X,\ d(x) \mapsto \{a(x)\}$$

From these definitions, we can define the set of solutions of a CSP $\langle d, C\rangle$ as

$$sol(\langle d, C\rangle) \stackrel{\text{def}}{=} \{a \in con(d) \mid \forall c \in C,\ a \in c\}$$

This mathematical framework leads to a "generate-and-test" backtracking algorithm. It enumerates all the possible combinations of values in the variables' domains and tests for each assignment if the constraint set is satisfied. Modern constraint solvers offer many optimizations building on this backtracking algorithm. To start with, we consider the "propagate-and-search" algorithm which interleaves two key steps:

Algorithm 1 Propagate-and-search `solve`($\langle d, C\rangle$)

Input: A CSP $\langle d, C\rangle$
Output: The set of all the solutions
1: $d' \leftarrow$ `propagate`($\langle d, C\rangle$)
2: **if** $d' = \emptyset$ **then**
3: **return** $\emptyset$
4: **else if** $d' = \{a\}$ **then**
5: **return** $\{a\}$
6: **else**
7: $\langle d_1, \ldots, d_n\rangle \leftarrow$ `branch`(d')
8: **return** $\bigcup_{i=0}^{n}$ `solve`($\langle d_i, C\rangle$)
9: **end if**

1. *Propagation* as an inference mechanism for removing values from the variables' domains that do not satisfy at least one constraint.
2. *Search* successively enumerates the different values in the variables' domains. The domains are backtracked to another choice if the former did not lead to a solution.

We define more precisely these two components and then give the full algorithm.

We see propagation abstractly as a function $propagate(\langle d, C\rangle) \mapsto d'$ mapping a CSP $\langle d, C\rangle$ to a reduced domain d' which is empty if the problem is unsatisfiable. We spend some times explaining propagation in more depth thereafter, but we give a first example of the propagation mechanism.

Example 1.1 (Propagation of $x > y$)**.** Consider the following CSP

$$\langle \{x \mapsto \{0,1,2\}, y \mapsto \{0,1,2\}\}, \{x > y\}\rangle$$

where x and y take values in the set $\{0,1,2\}$ and are subject to the constraint $x > y$. Without enumerating any value, we can already remove 0 from the domain of x and 2 from the domain of y since there is no solution with such partial assignment. Applying *propagate* to this problem maps to the domain function:

$$\{x \mapsto \{1,2\}, y \mapsto \{0,1\}\}$$

⌟

The second component, search, relies on a branching function splitting the domain into several complementary sub-domains.

Definition 1.3 (Branching function)**.** *The function $branch(d) \mapsto (d_1, \ldots, d_n)$ maps a domain function d to a sequence of domain functions $(d_1, \ldots, d_n)$ such that:*

(i) $con(d) = \bigcup_{i \in [1..n]} con(d_i)$ *(complete)*
(ii) $\forall i \in [1..n],\ con(d_i) \subset con(d)$ *(strictly monotonic)*

The two conditions ensure that splitting a domain must not remove some potential solutions (completeness) and must strictly reduce the domain set (strict monotonicity). This last condition is necessary to ensure the termination of the enumeration algorithm.

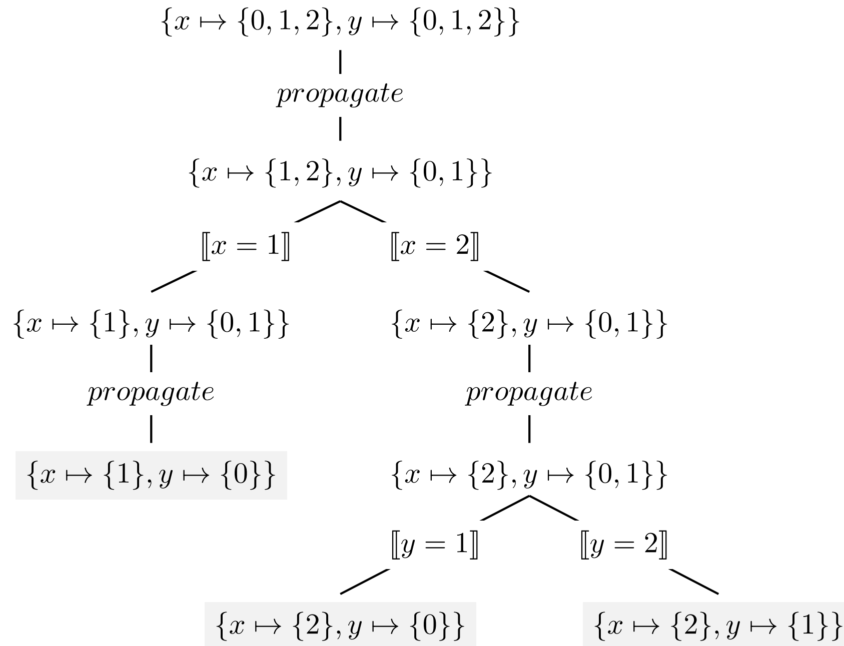


Figure 1.1: Search tree generated by the propagate-and-search algorithm.

Propagation and search are assembled in a backtracking algorithm presented in Algorithm 1. Whenever we obtain an empty domain d', we return an empty solution's set indicating that the input CSP has no solution. If the domain d' is an assignment after propagation, it ensures that it is a solution and we return it. The successive interleaving of choices and backtracks leads to the construction of a search tree. All in all, a search tree is obtained by recursively applying the branching function on the CSP, and by propagating the domains in each node.

Example 1.2 (Search tree of $x > y$)**.** In Figure 1.1, we unroll the propagate-and-search algorithm on the CSP given in the Example 1.1. It alternates between propagation and branching until we reach a leaf of the search tree which are all solutions (highlighted in grey). In this case, the branching function is a flat enumeration of the values in the domains, we will see other branching strategies in Section 1.5.3. ⌟

Constraint optimization problems

One of the most important variations of the propagate-and-search algorithm is for solving constraint optimization problems. Instead of finding one or all solutions, we

want to find the *best* solution according to an objective function. More specifically, given an objective function $f(x)$ where x is a variable, it tries to find a solution in which the value of x minimizes or maximizes $f(x)$. The usual algorithm is *branch and bound*: every time we reach a solution, we constrain the CSP such that the next solution is forced to be better (in the sense of the objective function). In branch and bound, the objective function is implemented with a constraint that is added globally to the CSP during the search. We delay the explanation of a more formal definition to Section 1.6.

1.1.3 Propagation

In the former definitions, constraints are given in an extensive manner but this is poorly suited for implementation purposes. To improve efficiency, we rely on the notion of *propagator*, which is a function representing a constraint. The role of a propagator is twofold: pruning the domain and deciding if an assignment is valid with regard to its induced constraint.

Definition 1.4 (Propagator)**.** *Let* $Dom \stackrel{\text{def}}{=} \mathcal{P}(V)^X$ *be the set of all the possible domain functions. Given a function* $p : Dom \to Dom$*, we say that* $a \in Asn$ *is a solution of* p *if* $p(dom(a)) = dom(a)$*. Then,* p *is a propagator if it satisfies the following properties:*

$$\begin{array}{ll} (P1)\ \forall d \in Dom,\ p(d) \subseteq d & \textit{(contracting)} \\ (P2)\ \textit{For any solution } a \textit{ of } p \textit{ and } \forall d \in Dom, & \textit{(sound)} \\ \quad dom(a) \subseteq d \textit{ implies } dom(a) \subseteq p(d) & \end{array}$$

where the inclusion between two domain functions $d \subseteq d'$ *is defined if* $\forall x \in X,\ d(x) \subseteq d'(x)$*.*

The contracting property ensures that a propagator p can only reduce domains and the soundness ensures that p cannot reject valid assignments. A constraint $c_p \in C$ is induced by a propagator $p \in Prop$ if $c_p = \{a \in Asn \mid p(dom(a)) = dom(a)\}$, i.e. the set of all solutions of p, where $Prop$ is the set of all propagators. Therefore, for a single constraint, there are several propagators that are more or less contracting but implement the same constraint. In the following, we use the function $[\![c]\!]$ to denote a propagator function for a constraint c, e.g. $[\![x > y]\!]$. We stay abstract over the exact definition of a propagator. The fact that it induces its constraint is generally enough.

Definition 1.5 (Propagation problem)**.** *A propagation problem* $\langle d, P\rangle$ *where* d *is the domain function and* $P \subseteq Prop$ *is the set of propagators is equivalent to the CSP:*

$$\langle d, \{c_p \in C \mid p \in P\}\rangle$$

where C *is a constraints set (Definition 1.2).*

The propagate-and-search algorithm can be directly adapted to deal with a propagation problem.

Intuitively, the function $propagate(\langle d, P\rangle)$ reduces the domain d by computing a fixed point of $p_1(p_2(\ldots p_n(d)))$ for every $p_i \in P$. Reaching a fixed point indicates that the propagators cannot infer additional information anymore and that we need to branch for progressing. This fixed point computation can be adequately formalized by viewing propagation as a transition system [Tac09].

Definition 1.6 (Transition system)**.** *A transition system is a pair $\langle S, \rightarrow\rangle$ where S is the set of states and $\rightarrow$ the set of transitions between these states.*

Definition 1.7 (Propagation-based transition system)**.** *The transition system of a propagation problem is a pair $\langle Dom, \rightarrow\rangle$ where the transition $\rightarrow: Dom \times Prop \times Dom$ models the application of a propagator to the domain. We require that any transition $(d, p, d') \in\rightarrow$, noted $d \xrightarrow{p} d'$, satisfies the following properties:*

$$\begin{array}{ll} (P1)\ p(d) = d' & \textit{(propagating)} \\ (P2)\ dom(p(d)) \subset dom(d) & \textit{(strictly monotonic)} \end{array}$$

The transition is lifted to CSP structures $\langle d, P\rangle \xrightarrow{p} \langle d', P\rangle$ with the additional condition that $p \in P$. We write its transitive closure as $\langle d, P\rangle \Rightarrow \langle d', P\rangle$ where $\langle d', P\rangle$ is necessarily a fixed point of the transition system.

It forms a non-deterministic transition system: there are more than one possible transition that can be applied on some states. Hence, there exists various algorithms for deciding in which order to apply the transitions in order to reach a fixed point faster. For example, we refer to [Tac09, PLDJ14] for more information on the various propagation algorithms. The following abstract definition of propagation will be sufficient in this dissertation.

Definition 1.8 (Propagation)**.** *Let $\langle d, P\rangle$ be a propagation problem, the propagation function is defined as*

$$propagate(\langle d, P\rangle) \mapsto \langle d', P\rangle \qquad \textit{where } \langle d, P\rangle \Rightarrow \langle d', P\rangle$$

Lastly, we give an important property of the propagation-based transition system.

Theorem 1.1 (Confluence)**.** *The fixed point of the transition system is unique if all the propagators are contracting, sound and monotonic (i.e. $d \subseteq d' \Rightarrow p(d) \subseteq p(d')$). Hence, the transition system is confluent.*

The proof can be found in [Tac09].

1.2 Lattice structure

We first recall some definitions on complete lattices and introduce some notations[2].

Definition 1.9 (Orders)**.** *An ordered set is a pair $\langle P, \leq \rangle$ where P is a set equipped with a binary relation $\leq$, called the order of P, such that for all $x, y, z \in P$, $\leq$ satisfies the following properties:*

$$(P1) \qquad x \leq x \qquad \textit{(reflexivity)}$$

$$(P2) \qquad x \leq y \wedge y \leq x \implies x = y \qquad \textit{(antisymmetry)}$$

$$(P3) \qquad x \leq y \wedge y \leq z \implies x \leq z \qquad \textit{(transitivity)}$$

$$(P4) \qquad x \leq y \vee y \leq x \qquad \textit{(linearity)}$$

An order is partial if the relation $\leq$ only satisfies the properties P1-P3, that is, it is not defined for every pair of elements. In this case, we refer to the set P as a partially ordered set (poset). Otherwise, the order is total and the set is said to be totally ordered.

In case of ambiguity, we refer to the ordering of the set P as $\leq_P$ and similarly for any operation defined on P. We can classify functions with several properties when applied to ordered sets.

Definition 1.10. *Given the ordered sets P and Q, and $x, y \in P$, a function $f : P \to Q$ is said to be order-preserving (or monotonic) if $x \leq_P y \Rightarrow f(x) \leq_Q f(y)$, an order-embedding if in addition the function is injective and an order-isomorphism if the function is bijective.*

We will focus on a particular class of ordered structures, namely lattices, and we introduce several definitions beforehand.

Definition 1.11 (Upper and lower bounds)**.** *Let $\langle L, \leq \rangle$ be an ordered set and $S \subseteq L$. Then*

- *$x \in L$ is a lower bound of S if $\forall y \in S,\ x \leq y$,*
- *S^ℓ denotes the set of all the lower bounds of S,*
- *$x \in L$ is the greatest lower bound of S if $\forall y \in S^\ell,\ x \geq y$.*

The (least) upper bound and the set of all upper bounds S^u are defined dually by reversing the order (using $\geq$ instead of $\leq$ in the definitions and vice-versa).

Definition 1.12 (Lattice)**.** *An ordered set $\langle L, \leq \rangle$ is a lattice if every pair of elements $x, y \in L$ has both a least upper bound and a greatest lower bound.*

[2] See for example [DP02] for a more in depth presentation of the order and lattice theories.

Definition 1.13 (Complete lattice)**.** *A lattice $\langle L, \leq \rangle$ is complete if every subset of $S \subseteq L$ has both a least upper bound and a greatest lower bound. A complete lattice is always bounded: there is a supremum $\top \in L$ such that $\forall x \in L,\ x \leq \top$ and an infimum $\bot \in L$ such that $\forall x \in L,\ \bot \leq x$.*

As a matter of convenience and when no ambiguity arises, we simply write L instead of $\langle L, \leq \rangle$ when referring to ordered structures. An alternative presentation is to view a lattice as an algebraic structure.

Definition 1.14 (Lattice as algebraic structure)**.** *A lattice is an algebraic structure $\langle L, \sqcup, \sqcap \rangle$ where the binary operation $\sqcup$ is called the join and $\sqcap$ the meet. The join $x \sqcup y$ is the least upper bound of the set $\{x, y\}$ and the meet $x \sqcap y$ is its greatest lower bound. We use the notation $\bigsqcup S$ (resp. $\bigsqcap S$) to compute the least upper bound (resp. greatest lower bound) of the set S.*

As shown in [DP02], the operations $\sqcup$ and $\sqcap$ satisfy several algebraic laws.

Theorem 1.2. *Let L be a lattice and $x, y, z \in L$. Then*

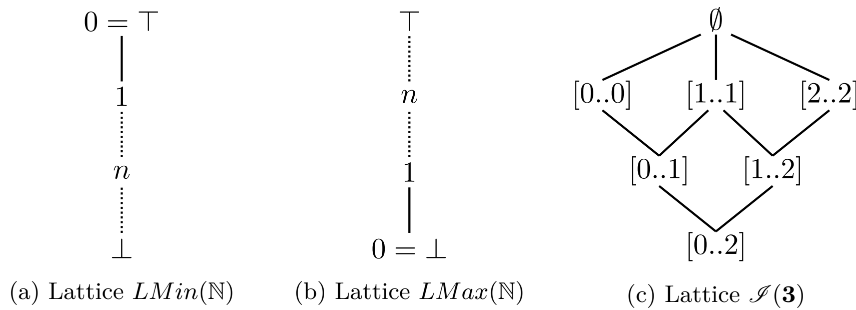
$$\begin{array}{ll} (L1)\ x \sqcup (y \sqcup z) \equiv (x \sqcup y) \sqcup z & (\textit{associative law}) \\ (L2)\ x \sqcup y \equiv y \sqcup x & (\textit{commutative law}) \\ (L3)\ x \sqcup x \equiv x & (\textit{idempotent law}) \\ (L4)\ x \sqcup (x \sqcap y) \equiv x & (\textit{absorption law}) \end{array}$$

We obtain the dual laws by exchanging $\sqcup$ with $\sqcap$ and vice-versa.

The distinction between lattices and complete lattices only arises for infinite sets since any finite lattice is complete [DP02]. The proof of this claim is given by unfolding an expression $\bigsqcup S$ as an expression $\forall s_i \in S,\ s_1 \sqcup \ldots \sqcup s_n$, by application of the associativity law 1.2(L1), we obtain a unique element. In the following, we consider lattices to be finite as it is usually the case in the field of constraint programming. Extending the framework developed in this chapter to infinite lattices is of great interest but is left to future works.
To complete this short introduction, we define the notion of sublattice which is a subset of a lattice that is itself a lattice.

Definition 1.15 (Sublattice)**.** *Let $\langle L, \leq \rangle$ be a lattice and $S \subseteq L$. Then S is a sublattice of L if for each $x, y \in S$ we have $x \sqcup_L y \in S$ and $x \sqcap_L y \in S$.*

In the following, we interpret the order of a lattice $\leq$ in the sense of information systems. It views the element bottom $\bot$ as being the element with the fewer information and the element top $\top$ as the element that contains all the information. This "contains more information than" order is noted $\models$ and named the *entailment* operator. The entailment relation is often the inverse of the intuitive order of a lattice. This has no impact on the mathematical ground since, by the duality principle, any theorem on a (complete) lattice is also valid for its inverse order. Note that our usage of the symbol $\models$ is not directly relevant to logical consequence and should be understood only as an order symbol.



(a) Lattice $LMin(\mathbb{N})$ (b) Lattice $LMax(\mathbb{N})$ (c) Lattice $\mathscr{I}(\mathbf{3})$

Figure 1.2: Examples of lattices' diagrams.

Notation 1.1 (Defining new lattice)**.** We define a new lattice as $L = \langle S, \models \rangle$ where S is its set of elements and $\models$ the order. Sometimes, the set S is already a lattice by construction, for example the Cartesian product of two lattices is a lattice. In this case, we omit the order which is inherited from S and we write $L = S$. When necessary, because we often manipulate lattices defined over other lattices, we disambiguate the order of a lattice with $\models_L$ where $\models$ is the order of L.

Example 1.3 ($LMin$ and $LMax$ lattices)**.** The lattices of decreasing and increasing natural numbers are defined as $LMin = \langle \mathbb{N}, \geq \rangle$ (Figure 1.2a) and $LMax = \langle \mathbb{N}, \leq \rangle$ (Figure 1.2b). Algebraically, we have $\langle LMin, min, max \rangle$ where the minimum operation between two numbers is the join and the maximum is the meet ; this is reversed for $\langle LMax, max, min \rangle$. For example, these lattices are used in the context of the $Bloom^L$ language for distributed programming [CMA$^+$12]. We use them extensively across this dissertation. ⌟

Example 1.4 (Lattice of natural intervals)**.** An natural interval with lower and upper bounds $l, u \in \mathbb{N}$ is defined as $[l..u] \stackrel{\text{def}}{=} \{x \mid l \leq x \leq u\}$. The set of natural intervals is defined as $\mathscr{I} = \{[l..u] \mid \forall l, u \in \mathbb{N}\}$. The lattice of natural intervals is $\langle \mathscr{I}, \models \rangle$ where $\models$ is the set relation $\subseteq$, the bottom element is the set $\mathbb{N}$ and the top element is the empty set $\emptyset$. This is illustrated in Figure 1.2c with a sublattice of $\mathscr{I}$ ranging over the numbers in $\mathbf{3} = \{0, 1, 2\}$. For example, the interval [0..0] contains more information than [0..2]. It captures the concept of a CSP's variable: we declare a variable with a domain between 0 and 2 because we do not know its exact value. ⌟

Deriving new lattices

In the Example 1.3, we made the choice of defining the lattices $LMin$ and $LMax$ over the set of natural numbers $\mathbb{N}$ but it could be equally defined for the set of integers $\mathbb{Z}$. More generally, as long as the underlying set is totally ordered, the corresponding $LMin$ and $LMax$ lattices can be defined. This hints to a notion of parametric structures where a lattice is built upon existing structures satisfying

some conditions, e.g. that the underlying set is totally ordered. In this chapter, we build a hierarchy of lattices and therefore it is worthwhile to introduce some notations.

Notation 1.2 (Parametric structure)**.** A parametric structure S relies on a sequence of structures $(P_1, \ldots, P_n)$. We note it $S(P_1, \ldots, P_n)$ and S is said to be a constructor. The arguments of a parametric structure are sometimes left implicit in a definition, in which case we suppose they satisfy the conditions given in the definition of S.

Using this notation we can rework the definitions of $LMin$ and $LMax$ to be more general.

Definition 1.16 ($LMin$ and $LMax$ lattices)**.** *Let $\langle S, \leq \rangle$ be a total order. Then*

$$LMin(\langle S, \leq \rangle) = \langle S, \geq \rangle$$
$$LMax(\langle S, \leq \rangle) = \langle S, \leq \rangle$$

Hence we write $LMin(\langle \mathbb{N}, \geq \rangle)$ for the decreasing lattice of natural numbers. For clarity, we will leave the order implicit, as in $LMin(\mathbb{N})$, when no ambiguity arises.

We define several other useful derivations that will be used across this dissertation. A basic but convenient derivation is to create a lattice—called a flat lattice—from any unordered set.

Definition 1.17 (Flat lattice)**.** *Any set S can be turned into a flat lattice by adding the two elements $\bot$ and $\top$ to S, and with the order $\forall x \in S,\ \bot \leq x \leq \top$.*

In computer science, this is extremely useful for embedding any set of values (described by a type) into a lattice.

Example 1.5 (Logic variable)**.** Logic variables pertain to languages based on logic programming. This concept dates back to the first implementation of Prolog and, more generally, is unavoidable when using unification. Indeed, the concept of unbound variable corresponds to a variable equals to $\bot$ and a bound variable to a variable equals to one of the values in the set S. As induced by the flat lattice's order, an unbound variable may become bounded at some points but this does not happen in the other direction if the program is monotonic. Trying to assign a new value to an already bounded variable results in the $\top$ element—equivalent to a failure—which possibly generates the backtracking of the system. Logic variable is one of the important concepts inherited by multi-paradigm languages supporting logic programming such as Oz [VRBD+03]. ⌟

We can compose two lattices to obtain a new one in different ways, we introduce the disjoint union and the linear sum operators (which are defined more generally on ordered sets). The linear sum $P \oplus Q$ is a more general derivation than the flat lattice in which every element in Q becomes greater than any elements in P.

Definition 1.18 (Linear sum)**.** *Given two disjoint posets P and Q, we define their linear sum as:*

$$P \oplus Q = \langle P \cup Q, x \models y \ \textit{if} \begin{cases} & x, y \in P \wedge x \models_P y \\ \vee & x, y \in Q \wedge x \models_Q y \\ \vee & x \in P, y \in Q \end{cases} \rangle$$

Given a poset S, its flat lattice construction is given by the linear sum $\{\top_S\} \oplus S \oplus \{\bot_S\}$ where $\bot_S, \top_S \notin S$. The disjoint union $P \dot\cup Q$ composes the ordered sets P and Q such that every element in the new set is still ordered as in their original set.

Definition 1.19 (Disjoint union)**.** *Given two disjoint posets P and Q, it is defined as follows:*

$$P \dot\cup Q = \langle P \cup Q, x \models y \ \textit{if} \begin{cases} & x, y \in P \wedge x \models_P y \\ \vee & x, y \in Q \wedge x \models_Q y \end{cases} \rangle$$

Schematically, the diagrams of P and Q are put side by side (for the linear sum one is put one below the other).

Cartesian product

The Cartesian product of two lattices gives a lattice with a coordinate-wise order.

Definition 1.20 (Cartesian product)**.** *Let L and K be two lattices. The Cartesian product $L \times K$ produces the following lattice:*

$$L \times K = \langle \{(x, y) \mid x \in L, y \in K\}, (x_1, y_1) \models (x_2, y_2) \ \textit{if} \begin{cases} & x_1 \models_L x_2 \\ \wedge & y_1 \models_K y_2 \end{cases} \rangle$$

The join and meet operations are also defined coordinate-wise.

Given the lattice $L_1 \times L_2$, it is useful to define the following projection functions, for $i \in \{1, 2\}$:

$$\pi_i : L_1 \times L_2 \to L_i \text{ defined as } \pi_i((x_1, x_2)) \mapsto x_i$$

An interesting property is that, given two monotone functions $f : L_1 \to L_1$ and $g : L_2 \to L_2$, and an element $x \in L_1 \times L_2$, we have

$$((f \circ \pi_1)(x), (g \circ \pi_2)(x)) \models x$$

That is, applying the functions f and g independently on each projection of x preserves the order of the structure. The proof of this claim is directly given by

definition of the order of the Cartesian product. For the sake of readability, we also extend the projection over any subset $S \subseteq L_1 \times L_2$ as:

$$\pi'_i : \mathcal{P}(L_1 \times L_2) \to \mathcal{P}(L_i) \text{ defined as } \pi'_i(S) \mapsto \{\pi_i(x) \mid x \in S\}$$

This derivation is the basis of compound data structures in programming languages such as structures in the imperative paradigm and class attributes in the object-oriented paradigm. In particular, the latest property allows us to prove that a program monotone over some components of a structure is monotone over the whole structure (which is a product of its components).

Using this derivation, we can generalize the lattice of natural intervals (Example 1.4) to arbitrary totally ordered sets using the $LMin$ and $LMax$ derivations.

Definition 1.21 (Lattice of intervals)**.** *Let S be a totally ordered set. The lattice of intervals of S is defined as:*

$$\mathscr{I}(S) = \{\top\} \cup \{(l, u) \in LMax(S) \times LMin(S) \mid l \leq_S u\}$$

The top element represent any interval where $l > u$.

Compared to the former definition, the usage of derivations allows for a more general formulation. The order is directly inherited from the Cartesian product which, in turn, uses the orders of $LMin$ and $LMax$. In the context of $\mathscr{I}$, we refer to the projection functions π_1 and π_2 respectively as *lb* and *ub* (standing for lower and upper bound).

Powerset-based derivations

We focus on an important family of derivations based on the powerset of a set. This is especially important for defining the lattice framework for constraint programming. Given any set S, we write $\emptyset \neq \mathfrak{L} \subseteq \mathcal{P}(S)$ a family of sets of S. If $\langle \mathfrak{L}, \subseteq \rangle$ is a lattice ordered by set inclusion it is known as the *lattice of sets* and we have the join as the set union and the meet as the set intersection. In our settings, this lattice is not so interesting because it does not rely on the order of the internal set S but instead sees its elements as atomic.

Example 1.6. Let $\langle \mathcal{P}(\mathscr{I}(\mathbf{3})), \models \rangle$ be the powerset lattice of the lattice $\mathscr{I}(\mathbf{3})$ given in Figure 1.2c and $S \models Q \iff S \subseteq Q$ be the order. Then

$$\{[0..0]\} \models \{[0..0], [1..1]\}$$

but $\{[0..0]\}$ and $\{[0..1]\}$ are unordered since $[0..0] \notin \{[0..1]\}$. ⌟

Of course, we would like to have $\{[0..0]\} \models \{[0..1]\}$ where $\models$ is defined inductively using $[0..0] \models_{\mathscr{I}(\mathbf{3})} [0..1]$. This leads us to a preliminary attempt by defining the order of the powerset of any lattice L as follows:

$$\langle \mathcal{P}(L), S \models Q \text{ if } \forall y \in Q,\ \exists x \in S,\ x \models_L y \rangle$$

Intuitively, a set S contains more information than Q if for each element $y \in Q$, there exists an element in S that contains more information than y. Unfortunately, this is not an order over $\mathcal{P}(L)$ because the intuitive ordering $\models$ is irreflexive in this context. Consider the following example:

$$\text{given } a = \{[0..1], [0..0]\} \text{ and } b = \{[0..0]\} \text{ does } a \overset{?}{\models} b \text{ hold?}$$

According to the definition we have $a \models b$ and $b \models a$ but $a \neq b$ so the relation is not reflexive and thus it is not an order. This is due to the fact that an element in a lattice L has several equivalent representations in the powerset $\mathcal{P}(L)$. To see that, consider $a, b \in L$ and $a \sqcup b = a \wedge a \neq b$, then we have at least $\{a, b\}$ and $\{a\}$ in $\mathcal{P}(L)$ which contains the "same amount of information", and thus should be equal.

We solve this problem by considering a specific lattice of sets called the antichain lattice. First, we define the notions of chain and antichain.

Definition 1.22 (Chain and antichain)**.** *Let P be a poset and $S \subseteq P$. Then*

- *S is a* chain *in P if S is a totally ordered set, and*
- *S is an* antichain *in P if S only contains unordered elements:*

$$\forall a, b \in S,\ a \models b \Rightarrow a = b$$

Therefore, an antichain does not contain the redundant elements that bothered us with the former order. This leads us to consider the lattice of antichains $\mathscr{A}(L) \subseteq \mathcal{P}(L)$ [CL01, BV18].

Definition 1.23 (Antichain completion)**.** *Let P be a poset, the antichain completion of P is given by*

$$\begin{aligned} \mathscr{A}(P) = \langle & \\ & \{S \subseteq \mathcal{P}(P) \mid S \textit{ is an antichain in } P\}, \\ & S \models Q \textit{ if } \forall y \in Q,\ \exists x \in S,\ x \models_P y \rangle \end{aligned}$$

When the base set P is a lattice, we have the following theorem.

Theorem 1.3. *The antichain completion $\mathscr{A}(L)$ of a lattice L is a lattice.*

Proof. We give a proof by contradiction. For any two antichains $S, Q \in \mathscr{A}(L)$ we must have $S \sqcup Q \in \mathscr{A}(L)$. If $R = S \sqcup_{\mathscr{A}(L)} Q$ is not an antichain then there are two elements $a, b \in R$ such that $a \models b$. Since S and Q are antichains (by definition of $\mathscr{A}(L)$), we must have $a \in S$ and $b \in Q$ (or reversed which is not a problem since $\sqcup$ is commutative). Since L is a lattice, we have $c = a \sqcup_L b$ and $c \in L$. Thus, we can build a set $R' = R \setminus \{a, b\} \sqcup c$ such that $R' \models R$. Hence R is not the least upper bound of S and Q, and it contradicts the initial hypothesis. Therefore, R is an antichain since it cannot contain two ordered elements, and thus $R \in \mathscr{A}(L)$. □

Store derivation

The concept of store is central to computer science as we manipulate values through variables, or at a lower-level, through memory addresses. This notion of variables—in the sense of programming languages—allows us to have duplicate values with different locations. This is why we give a lattice derivation to build a store from any lattice of values. We first define the intermediate notion of indexed poset and then define the lattice of a store.

Definition 1.24 (Indexed poset)**.** *Given an index set Loc and a poset of values V, an indexed value—that we call a variable—is a pair (l, v) where $l \in Loc$ and $v \in V$. The indexed partially ordered set of V is defined as follows:*

$$Indexed(Loc, V) = \langle Loc \times V, (l, v) \models (l', v') \text{ if } l = l' \text{ and } v \models_V v' \rangle$$

The order is identical to the one of the poset V when two values have the same location. Given an element $x \in Indexed(Loc, V)$, we use the projection function $loc(x)$ and $value(x)$ for retrieving respectively the location and the value of x. An indexed poset is particularly useful to define the notion of collection of values, namely a store.

Definition 1.25 (Store)**.** *Given a lattice L and an index set Loc, a store is defined as the indexed powerset of L in which every element contains only distinct locations. It is defined as follows:*

$$Store(Loc, L) = \{S \in \mathscr{A}(Indexed(Loc, L)) \mid \forall x, y \in S,\ loc(x) = loc(y) \Rightarrow x = y\}$$

It inherits the order of the antichain lattice. Its bottom element $\perp$ is the empty set.

We can view an element $s \in Store(Loc, L)$ as a function $s : Loc' \to L$ mapping a location to its value: $s(l) = v$ iff $\exists (l, v) \in s$ where $Loc' \subseteq Loc$ is the subset of location in s. In the following, when omitted, we consider the index set Loc to be equal to a finite subset of $\mathbb{N}$.

Example 1.7. Given the store $SI = S(\mathscr{I}(\mathbf{3}))$ of the lattice of integer intervals (modulo 3), the stores $\{(0, \perp)\}$ and $\{(2, [0..1]), (3, \perp)\}$ are elements of SI but $\{(0, \perp), (1, [1..2]), (1, [0..2])\}$ is not since there are two variables at the location 1. Also, notice that the order of the store is not the set inclusion. Indeed, consider $s_1 = \{(0, [0..2])\}$ and $s_2 = \{(0, [1..1]), (1, [0..1])\}$, $s_2 \models s_1$ is not defined under the set inclusion order, but it is intuitive that the store s_2 contains more information than s_1 since we have $[1..1] \models [0..2]$ and s_1 does not contain elements that are not in s_2. The order of the store ensures that $s_2 \models s_1$ holds whenever the values in s_2 are stronger than all the values in s_1. ⌟

We now show that this derivation is a lattice if the base structure is a lattice.

Lemma 1.4. *If L is a lattice, then the top element $\top$ of its derivation $Store(Loc, L)$ is the set $\{(l, \top) \mid l \in Loc\}$.*

Theorem 1.5. *Let L be a lattice. Then the store derivation $Store(Loc, L)$ is a lattice.*

Proof. We proceed by case analysis over the least upper bound of two stores $S, Q \in Store(Loc, L)$:

- If S equals $\top$, then $Q \sqcup \top = \top$ by definition of a lattice, and by Lemma 1.4 $\top \in Store(Loc, L)$.
- Similarly, if S equals $\bot$, then $Q \sqcup \bot = Q$ and from the hypothesis we have $Q \in Store(Loc, L)$.
- By 1.2(L2), it is similar with Q.
- For all pair of elements $(l, v) \in S$ and $(l', v') \in Q$ where $l = l'$, their joins, defined as $(l, v) \sqcup (l', v') = (l, v \sqcup_L v')$, belong to $Store(Loc, L)$. Indeed, since S and Q are antichains, there can only be two elements with the same location in $S \cup Q$, and since we join them, the result is unique in the new antichain. For the others elements with unique location in S or Q, they do not overlap in $S \sqcup Q$ and thus the result preserves the antichain property.

Hence, since every element S and Q has a least upper bound in *Store*, we conclude that *Store* is a lattice. □

Definition 1.26 (Store operations)**.** *Let Loc be a finite set of locations, L a lattice and $S = Store(Loc, L)$ a store. It is convenient to define several operations over a store:*

- *$minloc : S \to LMin(Loc)$ and $maxloc : S \to LMax(Loc)$ respectively maps to the minimal and maximal location of a value in a store. They are defined as follows:*

$$minloc(s) \mapsto min(\pi_1(s))$$
$$maxloc(s) \mapsto max(\pi_1(s))$$

- *$alloc : S \times L \to S$ for allocating a value in a store at a fresh location and is defined as*

$$alloc(s, v) \mapsto s' \text{ where } s' = s \sqcup \{(l, v)\} \text{ such that}$$
$$\forall l' \in Loc,\ l' > maxloc(s) \land l \geq l' \Rightarrow l = l'$$

The maximal location is used to allocate a value at the next available location.

Proposition 1.6. *The functions minloc, maxloc and alloc are surjective order-preserving functions.*

1.3 Lattice hierarchy for constraint programming

In Section 1.1, we saw that domains are defined as a set of values and constraints as a set of assignments. This hints to a hierarchical structure of the constraint framework where one layer relies on the lower layers. We formalize this intuition by defining these layers as lattice structures by successive powerset-based derivations. This is captured by the following notion of lattice hierarchy.

Definition 1.27 (Lattice hierarchy)**.** *Given a set S, a lattice hierarchy of S is a collection of lattices $(L_0, L_1, \ldots, L_n)$ such that:*

(i) *L_0 is the flat lattice of S.*

(ii) *For every lattice L_i with $i > 0$, we have a pair of functions $(h_{\downarrow_i}, h_{\uparrow_i})$:*

$$\begin{aligned} h_{\downarrow_i} &: L_i \to \mathit{Store}(L_{i-1}) \\ h_{\uparrow_i} &: L_{i-1} \to L_i \end{aligned}$$

where $h_{\downarrow_i}$ is an order-preserving function and $h_{\uparrow_i}$ is an order-embedding function.

On the one hand, the function $h_{\downarrow_i}$ shows a relation between a lattice L_i and the store derivation of its lower lattice L_{i-1}. It is not necessarily an embedding, and thus L_i might not be able to represent exactly every element in $\mathit{Store}(L_{i-1})$.On the other hand, $h_{\uparrow_i}$ represents a possible constructor from a value of L_{i-1} into L_i.

This concept allows us to give a classification of the structures manipulated in constraint solvers according to their levels in the hierarchy. We give a constraint hierarchy in Table 1.3 with several examples of constraint structures and programming languages specialized for particular layers. The function $h_{\uparrow_i}$ is given as a lattice derivation for each layer but should not be considered as definitive or unique. Indeed, as we later see in this chapter, many optimizations in constraint solvers are actually captured by more complex lattice derivations. However, the interesting point is that every layer represents a common concept across constraint solvers. Before going further and explaining every layer, it is worthwhile to give some notations and general properties over this hierarchy.

Notation 1.3 (Hierarchy numbering)**.** Given a lattice hierarchy H, the ith layer is noted $L_i = \langle S_i, \models_i \rangle$ where S_i is the underlying set and $\models_i$ the corresponding order. Similarly, the algebraic operations $\sqcup_i$ and $\sqcap_i$ are numbered as well as their set versions $\bigsqcup^i$ and $\bigsqcap^i$. Later on, we use a range notation i-j to refer to the layers between i and j, for example $\models_{0-2}$ refers to the set of orders $\{\models_0, \models_1, \models_2\}$.

Every lattice L_i in a hierarchy has a state space decomposition isolating a solution space Sol_i, a failed space Fail_i and a space $\mathit{Unknown}_i$ representing the set of elements that are neither solutions nor failures.

Level	L	Concept	Example
0	$\{\top\} \oplus S \oplus \{\bot\}$	Value	Logic variable
1	$\mathcal{P}(L_0)$, $\mathscr{I}(L_0)$	Domain	Set of intervals, bit array, intervals
2	$Store(L_1)$	Variable store	Array of variables
3	$L_2 \times \mathcal{P}(Prop)$	CSP	CHR [SVWSDK10], MiniZinc [NSB+07]
4	$\mathscr{A}(L_3)$	Search tree	Prolog, Oz [Sch02]
5	$Store(L_4)$	Tree store	Search combinators [STW+13]
		(Restart search)	Iterative deepening search [Kor85]
n	$Store(L_{n-1})$	Algorithm selection	EPS [PRS16], sunny-cp2 [AGM15]

Figure 1.3: Lattice hierarchy in the context of constraint programming.

Definition 1.28 (State space decomposition)**.** *The state space decomposition of a lattice L is a tuple of non-empty distinct posets* $(Sol, Fail, Unknown)$ *such that:*

$$L = Fail \ \oplus \ Sol \ \oplus \ Unknown$$

and by definition of $\oplus$ *we have* $\top_L \in Fail$ *and* $\bot_L \in Unknown$*. Given an element* $a \in L$*, we write* $Sol(a) = \{b \in Sol \mid b \models a\}$ *the solution space of a and similarly for its failed and unknown spaces.*

Intuitively, an unknown element can evolve to a solution or a failure, and a solution can evolve to a failure, but also to an unknown element. The latter case is possible, for example, when we add new variables into a store. For each lattice, we explicitly give its state space decomposition by specifying the solution and fail spaces. Given a lattice $\langle S, \models \rangle$, the identity

$$Unknown = (S \setminus Sol) \setminus Fail$$

is convenient to retrieve the unknown space when omitted.

1.3.1 L_0: Value

The smallest lattice in this hierarchy, in the sense of set cardinality, is the domain of discourse S_0 of the language lifted up to a flat lattice. In almost all cases, the set S_0 is a finite subset of the natural numbers $\mathbb{N}$, which is the one we consider in the examples.

Definition 1.29. *The state space decomposition of* L_0 *is*

$$(Sol = S_0 \setminus \{\top, \bot\}, Fail = \{\top\}, Unknown = \{\bot\})$$

1.3.2 L_1: Domain

The lattice derivation between L_0 and L_1 varies between solvers. Two mainstream derivations are the powerset $\mathcal{P}(L_0)$ and the interval lattice $\mathscr{I}(L_0)$ (Definition 1.21).

The first one is often implemented as a bit array when the domain is small to optimize the operations over domains (bit operators) and the memory consumption (for example, in the solver Minion [GJM06]). The second one can be advantageous when dealing with larger domains since only the lower and upper bounds need to be stored. A third possibility, lying in-between, is to use a set of intervals. It is isomorphic to $\mathcal{P}(L_0)$ while being more compact for larger or sparse domains. For all these derivations, the function $h_{\uparrow_1}$ is an order-embedding, and thus it is enough to ensure that any solution can be represented. Actually, even the identity function is a valid derivation, and thus L_0 and L_1 collapse into a single lattice—but it is inefficient since the only possible solving algorithm would be the enumeration of all the values.

The state space decomposition of L_1 in case of finite domain is given as follows.

Definition 1.30. *Given L_0 a finite domain, the state space decomposition of $\mathscr{I}(L_0)$ (and $\mathcal{P}(L_0)$) is given by:*

$$\begin{cases} Sol_1 = \{v \in L_1 \mid |v| = 1\} \\ Fail_1 = \{\top_1\} \end{cases}$$

The set of solutions is the set of assigned values: a domain of cardinality 1*. Note that $\top_1$ has the cardinality* 0*.*

A different state space decomposition can be defined when the domains are ranging over real numbers. In this case, the standard technique is to decide an element to be a solution when it is "small enough".

1.3.3 L_2: Variable store

The lattice L_2 is obtained with the order-embedding $Store(L_1)$ (Definition 1.25). It represents a set of variables mapped to a domain.

Definition 1.31. *The state space decomposition of L_2 is given by:*

$$\begin{cases} Sol_2 = \{a \in L_2 \mid \forall (l,v) \in a,\ v \in Sol_1\} \\ Fail_2 = \{d \in L_2 \mid \exists (l,v) \in d,\ v \in Fail_1\} \end{cases}$$

We have a solution when all the domains are assigned, and a failed store if one of the variable's domain is failed. Comparatively with the definitions in Section 1.1, the lattice L_2 corresponds to the set Dom and Sol_2 to the set of assignments Asn. Accordingly, we write an element $a \in L_2$ when it is an assignment—supposedly in Sol_2—and $d \in L_2$ otherwise.

1.3.4 L_3: Constraint satisfaction problem

The lattice L_3 represents the set of all CSPs. An element in L_3 is defined by the propagation problem $\langle d, P\rangle$ (Section 1.1). We will loosely speak of a CSP instead of a propagation problem.

Definition 1.32 (CSP)**.** *Let $Prop \subset L_2^{L_2}$ be the set of all propagators (Definition 1.4). Given a variable store L_2, the lattice L_3 of a CSP is defined as follows:*

$$\begin{aligned}&CSP(L_2) = \langle\\ &\quad L_2 \times \mathcal{P}(Prop),\\ &\quad \langle d, P\rangle \models \langle d', P'\rangle \text{ if } d \models_2 d' \land P' \subseteq P\rangle\end{aligned}$$

The set is obtained by the Cartesian product of the variable store L_2 and the powerset of the set of propagators $Prop$. In the following we call a set $P \subseteq Prop$ a constraint store.

The order $\langle d, P\rangle \models \langle d', P'\rangle$ indicates that either the domain has been contracted or that more propagators have been added into the problem. In either case, it formalizes the intuition that $\langle d, P\rangle$ contains more information than $\langle d', P'\rangle$.

Definition 1.33. *The state space decomposition of L_3 is given by:*

$$\begin{cases} Sol_3 = \{\langle d, P\rangle \in L_3 \mid \forall p \in P,\ \forall a \in Sol_2(d),\ p(a) = a\}\\ Fail_3 = \{\langle d, P\rangle \in L_3 \mid d \in Fail_2 \lor \exists p \in P,\ p(d) \in Fail_2\}\end{cases}$$

We observe a failure in L_3 if the value of a variable is failed—equals to $\top_1$—or if one propagator is failed—it maps to a value in $Fail_2$. Since the propagators in a solution are all entailed and do not bring much information anymore, we provide a convenient function that removes them and maps all the solutions of the problem in L_2:

$$Sol_{3to2}(\langle d, P\rangle) = \{a \in Sol_2(d') \mid \langle d', P'\rangle \in Sol_3(\langle d, P\rangle)\}$$

Finally, the hierarchical pair of functions $(h_{\downarrow_3}, h_{\uparrow_3})$ is defined as

$$\begin{aligned}&h_{\downarrow_3} : L_3 \to Store(L_2)\\ &h_{\downarrow_3}(\langle d, P\rangle) \mapsto \{(l, v) \mid l \in \mathbb{N}, v \in Sol_{3to2}(\langle d, P\rangle))\}\\ &h_{\uparrow_3} : L_2 \to L_3\\ &h_{\uparrow_3}(d) \mapsto \langle d, \emptyset\rangle\end{aligned}$$

$h_{\downarrow_3}$ is defined using Sol_{3to2} which actually corresponds to the definition in extension of constraints that was first introduced in Section 1.1.2. In other terms, it contains all the domains satisfied by every constraint. The upwards function $h_{\uparrow_3}$ wraps any domain d into a CSP with an empty set of propagators.

Transition system

We now relate the propagation's transition system introduced in Section 1.1 to the lattice L_3.

Definition 1.34 (Transition system)**.** *Given a lattice L_3, the transition system generated by an element $\langle d, P\rangle \in L_3$ is given by*

$$TS_3(\langle d, P\rangle) = \{\langle d', P'\rangle \mid \langle d, P\rangle \to^* \langle d', P'\rangle\}$$

where $\rightarrow^*$ is the propagation transition $\rightarrow$ applied an arbitrary number of times. TS_3 inherits the order of L_3. $\langle d, P\rangle$ is the bottom element and $\langle d', P'\rangle$ such that $\langle d, P\rangle \Rightarrow \langle d', P'\rangle$ is the top element.

Lemma 1.7. *Let $\langle d, P\rangle$ be a propagation problem. Then*

$$\langle d, P\rangle \rightarrow \langle d', P'\rangle \text{ implies } \langle d', P'\rangle \models_3 \langle d, P\rangle$$

Proof. First, by definition of the transition system, we have $P = P'$. Second, by the monotonicity property 1.7(P2), the transition $\rightarrow$ only generates domains d' such that $d' \subset d$, and thus $d' \models_2 d$. □

Lemma 1.8. *The transition system TS_3 has a unique top element.*

Proof. From the Lemma 1.7 and the confluence of the transition system, the top element is necessarily unique. □

This transition system coupled with the state space decomposition allows us to define the notion of convexity of a CSP.

Definition 1.35 (Convexity)**.** *A CSP $\langle d, P\rangle \in L_3$ is convex if the top element of its transition system $TS_3(\langle d, P\rangle)$ is an element of $Sol_3 \cup Fail_3$. A set of propagators P is convex if for each $d \in L_2$ the CSP $\langle d, P\rangle$ is convex.*

The non-convexity of a CSP formally implies that the propagation alone is not always sufficient to obtain the solution set of a CSP. This is why we need an upper layer L_4 to the lattice hierarchy. In the context of constraint programming, the problems are usually non-convex. We show a simple non-convex CSP below.

Example 1.8 (Non-convex CSP)**.** Consider the following CSP:

$$\begin{aligned} NeqXYZ = \langle \quad & \{x \mapsto [1..1], y \mapsto [1..3], z \mapsto [1..3]\}, \\ & \{[\![x \neq y]\!], [\![x \neq z]\!], [\![y \neq z]\!]\}\rangle \end{aligned}$$

A relevant subset of the lattice L_3 for *NeqXYZ* is presented in Figure 1.4. The three grey rectangles represent the transition systems inside the main lattice; for convenience, we annotate the relevant edges with the propagator applied to the domain. The transition system of *NeqXYZ* is non-convex since its top element is not in the solution space. We notice that two edges are not annotated by a propagator and not part of any transition system. Since the propagators reached a fixed point, we must increase the information into the CSP in order to reach the solution space. These edges—which are not transitions—are formalized with the lattice of the search tree L_4. ⌟

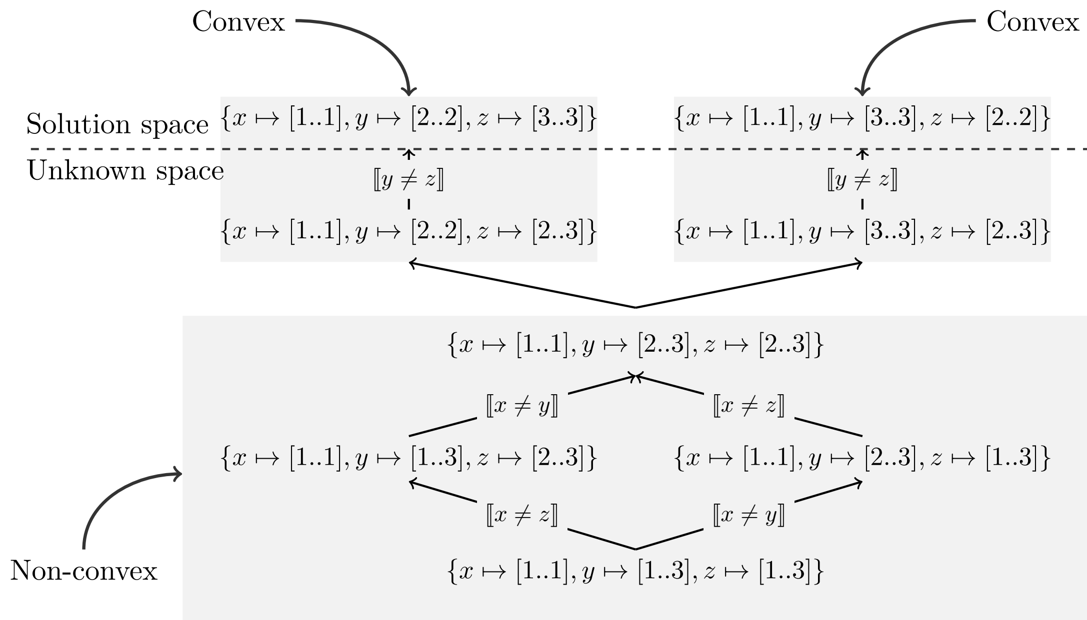


Figure 1.4: Example of a non-convex CSP: *NeqXYZ*.

1.3.5 L_4: Search tree

Considering a non-convex problem $\langle d, P\rangle \in L_3$, when reaching the top element of its transition system, we must make a choice by increasing the information in this problem, either by narrowing the variable store d or by adding a new constraint into the set P. We referred to this pattern in Section 1.1 as the "propagate-and-search" algorithm. In our lattice structure, the transition system is the propagation step and the search tree is created by increasing the information of the top element of this transition system. For example, in Figure 1.4, we see that connecting the grey zones form a search tree.

Intuitively, we represent a search tree with a set of nodes "to be explored"—namely the queue of nodes[3]—similarly to what is used in practice when exploring a tree or a graph. Schematically, the queue of nodes represents the leaves of a search tree.

Definition 1.36 (Queue of nodes)**.** *Given a lattice L_3, the lattice of queues of nodes L_4 is given by the antichain completion of L_3 written $\mathscr{A}(L_3)$.*

A queue of nodes is naturally represented as an antichain since the frontier of the search tree should only contain unordered nodes.

Definition 1.37. *The state space decomposition of the lattice L_4 is given as fol-*

[3]Despite the name, this terminology of "queue" does not imply a particular queueing strategy—i.e. the order in which the nodes are explored.

lows:

$$\begin{cases} Sol_4 = \{Q \in L_4 \mid \exists a \in Q,\ a \in Sol_3 \land \forall b \in Q,\ b \notin Unknown_3\} \\ Fail_4 = \{Q \in L_4 \mid \forall a \in Q,\ a \in Fail_3\} \end{cases}$$

The solution space is the set of queues such that each queue contains at least one solution node, and there is no node left to solve, i.e. in the unknown space. The failed space is the set of queues such that each queue contains only failed nodes.

The hierarchical pair of functions $(h_{\downarrow_4}, h_{\uparrow_4})$ is given by trivially ordering the nodes for $h_{\downarrow_4}$ and constructing a singleton node from L_3 for $h_{\uparrow_4}$.

Example 1.9. Consider the queues depicted in Figure 1.5; we can make the following observations:

- $Q_3 \models Q_1$ since Q_3 represents a more advanced exploration than Q_1 over the same set of nodes,
- Q_1 is not ordered with Q_2 since the state of exploration is different, but $Q_3 \models Q_2$ and $Q_3 = Q_1 \sqcup Q_2$.
- Since the second and third leftmost nodes are both in the queue, the queue Q_4 is not an antichain, and thus is not in L_4. In this case, we say that Q_4 is not compact.
- The queues Q_5 and Q_6 exhibit a search tree explored non-exhaustively. In Q_5, if there are some solutions in the rightmost subtree, we say that Q_5 is non-exhaustive. Similarly, Q_6 might be non-exhaustive to the search tree rooted at node R but is exhaustive relatively to A.

⌟

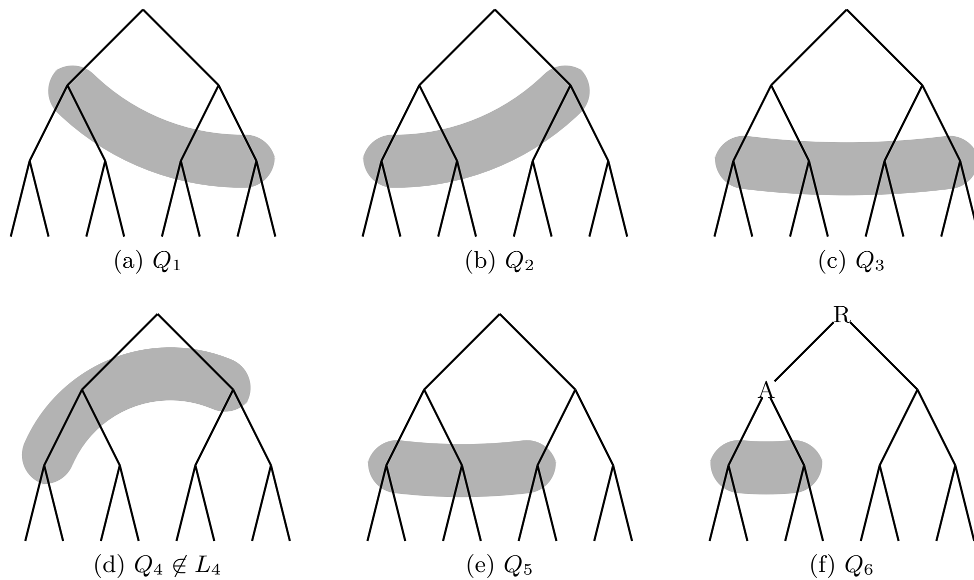
The concept of exhaustiveness is considered in more generalities in Section 1.5.4.

1.3.6 L_5: Tree store

L_5 is the lattice of all the possible ways to explore multiple search trees. This scheme occurs mainly with “restart-based search strategies” exploring a partial search tree and restarting the search by exploring another one. An element $\{(l_1, q_1), (l_2, q_2), \ldots, (l_n, q_n)\} \in L_5$ contains n search trees being explored where every $q_i \in L_4$ is a queue of nodes labelled by a location $l_i \in Loc$.

Definition 1.38 (Tree store)**.** *Given a lattice L_4, the lattice of all the possible sets of queues is given by the store derivation*

$$L_5 = Store(L_4)$$



(a) Q_1 (b) Q_2 (c) Q_3

(d) $Q_4 \notin L_4$ (e) Q_5 (f) Q_6

Figure 1.5: Examples of queues with different properties.

Indexing the queue is necessary because some search strategies re-explore the same search tree several times. Also, note that this lattice does not enforce sequentiality: given a sequence (q_1, q_2), the search trees q_1 and q_2 might be explored concurrently. This is for example required by interleaving search strategies [Mes97]. We give two examples of restart-based search strategies illustrating a strategy over L_5.

Example 1.10 (Iterative deepening depth-first search (IDS))**.** A very common restart strategy is IDS [Kor85] where a search tree is first explored at depth 1, then at depth 2, and until we explored the full search tree. This search strategy generates a set of queues $(q_1, \ldots, q_n)$ that exhaustively explores a problem. Note that the queue q_i subsumes the search tree of q_{i-1} with an extended layer of nodes. We say that IDS is not compact because a node (besides the root node) can be explored in more than one queue. ⌟

Example 1.11 (Limited discrepancy search (LDS))**.** LDS [HG95] is designed for problems where the strategy often goes straight to the goal. It explores a first search tree by taking only left branches, then restarts by allowing to take one right branch—a "wrong turn"—and so on until we explored the full search tree. Similarly to IDS, it is an exhaustive strategy that is not compact. Improved LDS (ILDS) [Kor96] is a variant of LDS that is exhaustive and compact. Indeed, it explores the nodes with exactly i discrepancies at the iteration i (and not with discrepancies $\leq i$ like in LDS). ⌟

1.3.7 L_n: Algorithm selection

Intuitively, it is always possible to define a level higher in the hierarchy by exploiting the store derivation. For example, a hypothetical lattice L_6 is the structure explored by several restart-based search strategies. However, since most solving algorithms are either defined over the lattices L_4 or L_5, we generalize the successive store derivation to the lattice L_n representing all the possible ways to solve a CSP.

Definition 1.39 (Lattice for algorithm selection)**.** *Given any lattice L_{n-1}, we can define the lattice $L_n = Store(L_{n-1})$.*

This lattice is well studied in artificial intelligence under the name of "algorithm selection" [Ric76]. Informally, given a collection of algorithms, this problem consists in selecting one algorithm that is efficient for a given problem or problem's instance. For example, SATzilla [XHHLB08] selects an algorithm among several SAT solvers, and sunny-cp2 [AGM15] among several constraint solvers. The key idea behind these algorithms is to learn from a set of problem's instances which algorithm is working best using machine learning methods. Another instance of the algorithm selection is in the context of parallel search—this specific case is called a "portfolio algorithm". Embarrassingly parallel search (EPS) [RRM13] is an algorithm that splits a problem into many sub-problems solvable independently by a parallel unit. Using this method, they design a parallel strategies selection algorithm [PRS16] testing a set of strategies on the sub-problems generated by the EPS algorithm. According to their efficiency on their respective sub-problems, the most efficient strategy is selected.

1.4 Inference in L_3 with propagation

Constraint inference is the process of deriving new information from existing knowledge without any guess. Since we do not make any guess, the inference is never wrong and never remove potential solution from the problem. We can perform inference at different level of the hierarchy: constraint propagation is a form of inference in L_3 and no-goods learning is another form of inference over L_4. We focus on the inference in L_3 first and come back on the inference in L_4 in Section 1.7. As a first step, we use the algebraic view of the lattice L_3 to model a constraint problem (Section 1.4.1). Following some ideas introduced by Saraswat with concurrent constraint programming [SR89], we show that the operators $\sqcup_{0-3}$ and $\models_{0-3}$ can be used for programming propagators. Using this framework, we consider the two roles of a propagator—namely deciding and propagating over L_3—respectively in the Sections 1.4.2 and 1.4.3.

1.4.1 Modelling a problem

The join operation $\sqcup$ allows us to add information; this operator is the only one needed to build the model of a CSP. In L_2, it is used for creating new variables

and in L_3 for adding new constraints. We consider the CSP

$$\langle \{x \mapsto [0..2], y \mapsto [1..2]\}, \{[\![x < y]\!], [\![x \neq 0]\!]\}\rangle$$

all along this section. The notation $[\![c]\!]$ maps the constraint c to a propagator (see Section 1.1). An initial and empty constraint model consists of each layer of the lattice hierarchy initialized to its bottom element. We consider the lattice of values L_0 where S_0 ranges over integers and the variable store L_2 to be indexed by the set of locations $Loc = \{x, y\}$.

Example 1.12 (Creating variables)**.** Given the empty variable store $\bot_2$, we add the variables x and y into the variable store $d \in L_2$ as follows:

$$d = \bot_2 \sqcup_2 \{(x, \{0, 1, 2\})\} \sqcup_2 \{(y, \{1, 2\})\}$$

⌟

This variable store enables us to build a CSP $R \in L_3$ and to add the constraints into R.

Example 1.13 (Adding constraints)**.** Using the domain $d \in L_2$ built in the previous example, we build the CSP as follows:

$$R = \bot_3 \sqcup_3 \langle d, \{[\![x < y]\!], [\![x \neq 0]\!]\}\rangle$$

⌟

The obtained CSP R is the model of our problem and the root node of a search tree. Constraint-based languages usually provide statements for the join operators $\sqcup_{0-3}$. Such modelling languages are analysed in Section 2.3.

1.4.2 Deciding

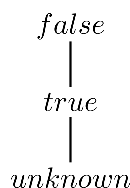


Figure 1.6: Lattice *ES* of the entailment status.

Inevitably, a propagation engine needs to check if a propagator p is entailed from the current domain d. More formally, it is described by the entailment relation $\langle d, \emptyset\rangle \models \langle d, \{p\}\rangle$. For short, we will write $d \models_d p$ where $\models_d$ is called the *domain entailment*. The domain entailment is used in many backtrack-free constraint-based languages; we review them in Section 2.4.

Example 1.14 (Removal of entailed propagators)**.** The domain entailment is a well-known optimization in constraint solvers. Every propagator $p \in P$ is marked when it is entailed by the domain d so it will not be tested again. The soundness property 1.4(L2) implies that a propagator is always either entailed or disentailed by the domain when all the variables are assigned. Hence, whenever all the propagators are annotated, it means we reached a solution. ⌟

In practice, we are interested in three complementary events revolving around the domain entailment:

1. The propagator is entailed: d is a solution of p.
2. The propagator is disentailed: d will never be a solution of p.
3. We do not know yet.

Operationally, the entailment relation is a boolean predicate, and thus it cannot serve as such for establishing the entailment status of a propagator. Interestingly, this set of events forms a complete lattice that we call ES, as depicted in Figure 1.6.

Definition 1.40 (Entailment status)**.** *The entailment status is the lattice* $ES = \langle\{false, true, unknown\}, false \models true \models unknown\rangle$.

This lattice can be used to characterize an element in its state space decomposition. Let L be a lattice and $(Sol, Fail, Unknown)$ its state space decomposition. Then there exists a function $ssd : L \to ES$ defined as:

$$ssd(a) \mapsto \begin{cases} true & \text{if } a \in Sol \\ false & \text{if } a \in Fail \\ unknown & \text{if } a \in Unknown \end{cases}$$

In addition, this function is monotonic if the lattice cardinality of the lattice L_2 is stable. The property that ssd is monotonic shows that an entailed or disentailed propagator will preserve its status for any function progressing in *Unknown*[4]. The domain entailment problem $d \models_d p$ can be defined as $ssd(\langle d, \{p\}\rangle) = true$.

The case where the domain d does not have a fixed cardinality is relevant to systems with infinite computations, and we leave their analysis to future work.

1.4.3 Propagating

We show that we can give an intentional description of a propagator using the entailment $\models$ and the join operator $\sqcup$ to infer information. First, we introduce the notion of event e between two domains.

[4]Generally, we stop the computation whenever we reach an element in *Sol* or *Fail*. However, note that an element in *Sol* can evolve to a failed state if further strengthened.

Definition 1.41 (Event relation)**.** *Given a lattice L and a relation $e : L \times L$, we say that e is an event relation of L if:*

$$\forall x, y \in L,\ e(x, y) \Rightarrow (x \models y \wedge x \neq y)$$

$x \models y$ ensures that the event e is monotonic over L and $x \neq y$ ensures that it describes a real change. Since the relation e is irreflexive, it is not an order.

Events are central in a propagation engine for scheduling the propagators only when it is relevant to do so, i.e. they will reduce the domain.

Example 1.15 (Event `lbc`)**.** Given a lattice L_1, the event `lbc` describes a change on the lower bound of a domain. Let $x, x' \in L_1$, then `lbc` is defined as

$$\texttt{lbc}(x, x') \text{ if } lb(x') \models_{LMax} lb(x) \wedge x \neq x'$$

Intuitively, it is read as "the lower bound of the updated domain x' contains more information than the one of the domain x". ⌟

The dual event `ubc` is defined similarly. With these two events, we can program a propagator for the constraint $x < y$.

Example 1.16 (Propagator $x < y$)**.** Consider two lattices $d, d' \in L_2$ where d is the current variable store and d' is the former one. For convenience, we note $x_{lb} \stackrel{\text{def}}{=} lb(d(x))$ and $x_{ub} \stackrel{\text{def}}{=} ub(d(x))$.

$$[\![x < y]\!] \mapsto \begin{cases} \text{if } \texttt{lbc}(d'(x), d(x)) & \text{then } d = d \sqcup_2 \{(y, [(y_{lb} \sqcup_{LMax} x_{lb} + 1)..y_{ub}])\} \\ \text{if } \texttt{ubc}(d'(y), d(y)) & \text{then } d = d \sqcup_2 \{(x, [x_{lb}..(x_{ub} \sqcup_{LMin} y_{ub} - 1)]\} \end{cases}$$

The propagator $[\![x < y]\!]$ is contracting because every operation applied on the lattices involved (L_1, L_2, $LMin$ and $LMax$) is monotonic. ⌟

We can mention the indexicals language [VHSD91] which allows users to build propagators using domains and monotonic operators. In Section 2.4.2, we give the implementation with indexicals of the same propagator $x < y$. The resemblance between the obtained program and our mathematical definition is striking.

1.5 Bridging L_3 and L_4 with backtracking

Backtracking is a central concept in computer science as it allows programmers to write non-deterministic algorithms. It is necessary when an algorithm needs to make a choice in order to progress. For example, it is necessary to solve a non-convex CSP. In this section, we argue that backtracking can be modelled with the meet operator $\sqcap$ because it triggers the removal of information. More precisely, the meet operator is used with a delta operator Δ for computing how much information we must retract (Section 1.5.1). This operator applied to L_2 and

L_3 is useful to restore the CSP to a former state when backtracking. In L_4, it is useful to extract a node from the queue but also to prune a part of the search tree as we see in Section 1.7. The operators $\sqcup$ and Δ are sufficient to create and explore the search tree of a CSP. Such a search tree forms a "raw material" that will be transformed and pruned according to inference and pruning strategies. We focus in this section on the creation and exploration of this raw search tree; it involves several sub-components that we briefly introduce with the following example.

Example 1.17 (Creating a search tree)**.** Given the CSP R—built in the Example 1.13—we initialise a queue of nodes to the singleton $\{R\} \in L_4$ where R is the root node of the search tree. The tree to be created is shown in Figure 1.7a. We create two new nodes $A = R \sqcup \langle \bot, \{[\![x < 2]\!]\}\rangle$ and $B = R \sqcup \langle \bot, \{[\![x \geq 2]\!]\}\rangle$. Given the queue q, we update it with $q = q \sqcup \{A, B\}$, that is, the node R is automatically absorbed by A and B and removed from the queue while A and B are pushed onto the queue. ⌟

There are several distinct actions happening in this example and we define them individually to obtain a framework closer to the solver's implementation. Specifically, this section is dedicated to the definition of an exploration strategy that is in charge of expanding nodes in a queue. The goal is to formalize the process behind expansion that we call an exploration strategy.

Intuitively, given a queue $q \in L_4$, the process of expanding a node is usually divided into the following actions:

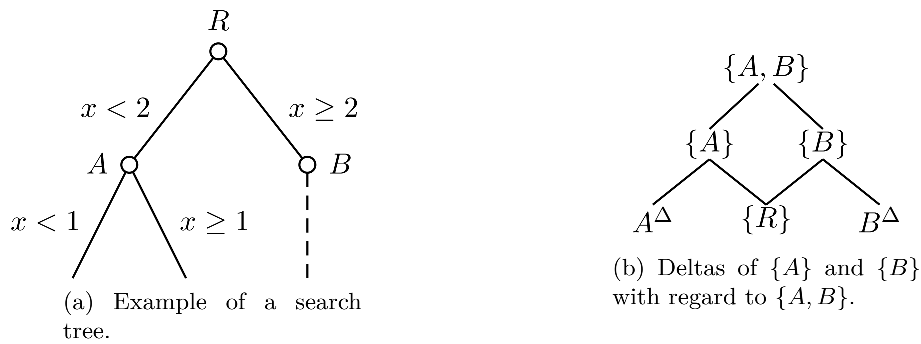
1. Selecting a node $a \in q$.
2. Splitting the node a into several children nodes $(a_1, \dots, a_n)$.
3. Inserting the sequence $(a_1, \dots, a_n)$ onto the queue.

The actions (1) and (3) form a queueing strategy while (2) is called the branching strategy.

More formally, this section is dedicated to solving a CSP by finding a fixed point of the following exploration strategy:

$$push \circ copy \circ branch' \circ propagate' \circ pop$$

The function *propagate'* encapsulates the propagation function of a CSP (Definition 1.8) to be compatible with the codomain of *pop* and the domain of *branch'*. The queueing strategy is specified by the pair of functions (*push*, *pop*) and is formalized in Section 1.5.2. Splitting the state space is realized by the functions *branch'* and *copy* which are introduced in Section 1.5.3. Thereafter, we study several properties of the exploration strategy as a whole in Section 1.5.4.



(a) Example of a search tree.

(b) Deltas of $\{A\}$ and $\{B\}$ with regard to $\{A, B\}$.

Figure 1.7: Backtracking in a search tree with the delta operator.

1.5.1 Delta operator

We mentioned that information can be removed using the meet operator $\sqcap$, for example to extract a node from a queue. Intuitively, given a queue q and a node $x \in q$, we would like to extract x from q with $q \sqcap y = q \setminus x$. Two questions are how to compute y and which one to pick if it is not unique? Substituting y with $q \setminus x$ is valid but it is not interesting from an algorithmic point of view. Therefore, we need another operator to compute a delta between two elements.

Definition 1.42 (Delta operator)**.** *Let L be a lattice and $a, b \in L$ such that $a \models b$. The deltas of b with regard to a is the set*

$$D = \{d \in L \mid b \sqcup d = a\}$$

$a\Delta b$ is the smallest element in D such that

$$a\Delta b = \bigsqcap D$$

In the theory of lattices, Δ is called the *weak relative pseudo-complement* [GPR96]. We chose the symbol Δ because it expresses the notion of difference which is what is important here.

Example 1.18 (Extracting a node)**.** Continuing with the Example 1.17 with the queue q, the search proceeds by extracting either the node A or B. In Figure 1.7b, we show the delta elements of the nodes A and B with regard to $\{A, B\}$ where

$$\begin{array}{ll} \text{(i)} & A^{\Delta} = \{B\}\Delta\{A, B\} = \langle \bot, \{[\![x < 2]\!]\}\rangle \\ \text{(ii)} & B^{\Delta} = \{A\}\Delta\{A, B\} = \langle \bot, \{[\![x \geq 2]\!]\}\rangle \end{array}$$

which correspond to the constraints on the relevant branches. Admitting we want to extract A, we use the delta operator to obtain the queue $\{B\}$ as follows:

$$\{B\} = q\Delta(q\Delta\{A\})$$

Applying twice Δ on the queue enables us to recover the queue without the information that was only contained in the node A. ⌟

The double Δ is only necessary if the order of the structure is defined according to the order of its inner elements. In our case, the order of L_4 depends on the one of L_3 (which is turn depends on L_2 and then L_1). It means that the delta element is the smallest possible, while in practice we would like to keep the information that overlaps with other nodes. If we do not, we would always remove at least the root node, and thus the problem's model, since it overlaps with every node. Therefore, we must apply a second time the delta operator over the queue to remove just the information that only belong to the node considered. In the following, we use a store derivation over L_4, and thus since the order of a store is designed to accept duplicate elements, a single delta will always suffice.

1.5.2 Queueing strategy

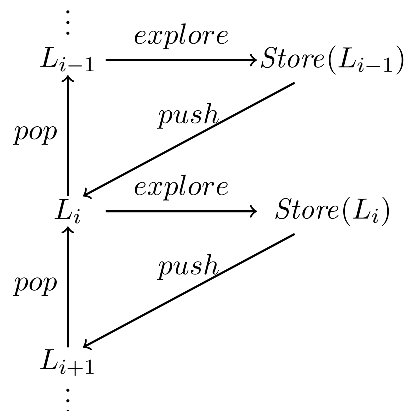


Figure 1.8: Abstract exploration strategy.

The exploration of a tree or a graph is implemented by mean of a data structure providing at least two operations: *push* and *pop* for respectively pushing the successors' nodes and extracting a node from the queue. Anticipating on what follows, the queueing strategy actually forms the basis of the exploration strategy as depicted in Figure 1.8. It provides a framework to describe an exploration strategy by traversing up and down the lattice hierarchy. Therefore, a queueing strategy is always defined on two successive lattices L_i and L_{i-1}. Moreover, queueing is intrinsically tied to sequencing nodes in a certain order. To achieve this, we provide a lattice derivation, based on the store derivation, indexing the element L_{i-1} of the structure L_i. For example, we need to index the nodes of the search tree in L_4 in order to explore them with a depth-first or breadth-first search strategy.

Definition 1.43 (Queuing derivation)**.** *Let L_i and L_{i-1} be two successive lattices in a hierarchy and Loc be an indexed set. Then, Q_i is the lattice of the indexed elements of L_i:*

$$Q_i = \{S \in Store(Loc, L_{i-1}) \mid \pi'_2(S) \in L_i\}$$

where π'_2 is the projection over every element in the set S that excludes the locations. The order is inherited from the Store derivation.

Importantly, the projection of an element in the lattice Q_i must belong to L_i.

Proposition 1.9. *L_i is embedded in Q_i.*

Proof. By definition of Q_i, L_i is necessarily embedded in Q_i because every element in L_i has several indexed counterparts in Q_i. □

A queueing strategy is then defined over a lattice Q_i as follows.

Definition 1.44 (Queueing strategy)**.** *Let L_{i-1} be a lattice and Q_i the queuing derivation of its successor. Consider the following pair of functions:*

$$\begin{aligned} &push : Q_i \times Store(L_{i-1}) \to Q_i \\ &pop : Q_i \to Q_i \times L_{i-1} \end{aligned}$$

Then $(pop, push)$ is a queueing strategy if for every $S \in Q_i$ and $b \in Store(L_{i-1})$ we have

$$\begin{aligned} &(P1)\ \pi'_2(S) = \pi'_2(S') \sqcup_i \{x\} \text{ where } (S', x) = pop(S) \\ &(P2)\ \pi'_2(push(S, b)) = \pi'_2(S) \sqcup_i \pi'_2(b) \end{aligned}$$

An important consideration is that the properties $(P1)$ and $(P2)$ are over L_i and not Q_i (we use the projection π'_2 to recover the element in L_i). This allows the queueing strategy to reorder the locations arbitrarily. For example, a priority queue reorders the nodes at every insertion, and thus their locations change.

We illustrate this derivation with two of the most standard queueing strategies: depth-first search (DFS) and breadth-first search (BFS). We consider the set of locations Loc to be a finite subset of $\mathbb{N}$ ordered by $\leq$.

Example 1.19 (Depth-first search)**.** Let $q \in Q_i$ be a queue of nodes and $b \in Store(L_{i-1})$ the nodes to insert in the queue. Then the DFS queueing strategy is given by

$$\begin{aligned} &(i) \quad pop(q) \mapsto (q\Delta_4\{(maxloc(q), v)\}, v) \text{ where } v = q(maxloc(q)) \\ &(ii) \quad push(q, b) \mapsto q \sqcup_4 \{(maxloc(q) + maxloc(b) + 1 - l, v) \mid (l, v) \in b\} \end{aligned}$$

The function *pop* always extracts the node at the top of the stack which is the one with the greatest location. We use the delta operator for extracting this node. The set of nodes b to push over q represent an ordered set of children nodes. By convention, we want the node with the lowest location in b to be popped first since it represents the left-most node. Hence, in order be generic over any store of nodes b, we use the expression $maxloc(q) + maxloc(b) + 1 - l$ for reversing the order of insertion and simulating a stack exploring the children from left-to-right. ⌟

Example 1.20 (Breadth-first search)**.** Similarly to DFS, we can define the BFS queueing strategy by

$$\begin{array}{ll} (i) & pop(q) \mapsto (q\Delta_4\{(minloc(q), v)\}, v) \text{ where } v = q(minloc(q)) \\ (ii) & push := push_{DFS} \end{array}$$

Similarly to a FIFO queue, we push a node at the "back of the queue" and extract a node "at the front of the queue"—the node with the smallest location. The duality with DFS is clear in the function *pop* since we only exchanged *maxloc* with *minloc*. Pushing a node is defined exactly as for DFS. ⌟

1.5.3 Branching strategy

In Section 1.1, we introduced the branching function $branch : L_2 \rightarrow \mathcal{P}(L_2)$ splitting a variable store into two or more stores. Actually, the branching function can also map to a set of constraints instead of directly pruning the variable store—which is what is implemented in practice. We give a definition generalizing these possibilities.

Definition 1.45 (Branching function)**.** *A branching function is a function* $branch : L_3 \rightarrow Store(L_3)$ *strictly monotone over* $Unknown_3$*. We have, for each* $x \in L_3$*,* $branch(x) = b$ *such that*

$$\begin{array}{lll} (i) & ssd(x) \neq unknown \Rightarrow b = \bot_{Store(L_3)} & \textit{(stop criterion)} \\ (ii) & \forall y \in \pi'_2(b),\ x \sqcup_3 y \models x \land x \sqcup_3 y \neq x & \textit{(strict monotonicity)} \end{array}$$

The first condition indicates that we only split the state space of the unknown space. Indeed once we obtain a solution or a failed node, there is usually no reason to further divide the state space. By mapping to an empty set, here $\bot_{Store(L_3)}$, no node is added onto the queue by the function *push*. The second condition forces the children nodes to contain more information than the initial node. We can refine (ii) into two additional properties: exhaustiveness and compactness. Intuitively, exhaustiveness ensures that all solutions are enumerated and compactness that they are enumerated at most once.

Property 1.1 (Exhaustiveness)**.** *Let branch be a branching function and* $x \in L_3$*, it is exhaustive if*

$$\forall d \in Sol_{2to3}(x),\ \exists y \in \pi'_2(branch(x)),\ \exists d' \in Sol_{2to3}(y),\ d' \models_2 d$$

Every solution of x *is also a solution of at least one of its child.*

Property 1.2 (Compactness)**.** *Let branch be a branching function and* $x \in L_3$*, it is compact if*

$$\forall y, z \in \pi'_2(branch(x)),\ \exists d \in Sol_{2to3}(y),\ \exists d' \in Sol_{2to3}(z),\ d \models_2 d' \Rightarrow y = z$$

In simpler terms, for any pair of children y, z *of a node* x*, their sets of solutions in* L_2 *do not overlap.*

Typically, we are interested in a specific class of branching functions defined over L_2—called the domain branching functions.

Definition 1.46 (Domain branching)**.** *A domain branching function is the composition of three functions defined as follows:*

- $var_order : L_2 \rightarrow Loc$ *selects a variable from the variable store.*
- $val_order : L_2 \times Loc \rightarrow L_0$ *selects a value in the domain of the variable chosen by* var_order*.*
- $distribute : Loc \times L_0 \rightarrow Store(L_3)$ *splits the CSP on the selected variable and value.*

Let $x \in L_3$*, we define the branching function as follows:*

$$\begin{array}{l} branch(x) \mapsto \bot_{Store(L_3)} \text{ if } ssd(x) \neq unknown \\ branch(\langle d, P \rangle) \mapsto distribute(l, val_order(d, l)) \text{ where } l = var_order(d) \end{array}$$

Example 1.21 (Input-order split middle branching)**.** A basic branching function is to take the variable in order (according to the order on *Loc*) and to split on the middle value.

$$\begin{cases} input_order(d) & \mapsto minloc(\{(l, v) \in d \mid |v| > 1\}) \\ middle_value(d, l) & \mapsto (lb(d(l)) + ub(d(l)))/2 \\ bisect(l, m) & \mapsto alloc(alloc(\bot_4, \langle \bot_2, [\![l < m]\!] \rangle), \langle \bot_2, [\![l \geq m]\!] \rangle) \end{cases}$$

⌟

For non-binary strategies such as plain enumeration all of the values, i.e. $x = v_1 \vee x = v_2 \vee ... \vee x = v_n$ for every v_i in the domain of x, the function *val_order* and *distribute* are merged.

Example 1.22 (Enumeration branching)**.** Consider any variable ordering function *var_order*, then we can define the enumeration branching strategy as follows:

$$\begin{array}{l} enumerate : L_2 \times Loc \rightarrow Store(L_3) \\ enumerate(d, l) \mapsto \{\langle \{l \mapsto v\}, \bot \rangle \mid v \in d(l) \wedge l = var_order(d)\} \end{array}$$

In this case, we chose to directly prune the domain of the CSP instead of adding constraints. ⌟

In order to be composed seamlessly with the queueing strategy, we embed the branching function with the current queue that is just forwarded from the domain to the codomain:

$$\begin{array}{l} branch' : Q_4 \times L_3 \rightarrow Q_4 \times L_3 \times Store(L_3) \\ branch'(q, x) \mapsto (q, x, branch(q)) \end{array}$$

As shown in the previous example, the branches only contain the deltas with regard to the current node. We use a copy restoration strategy for reconstructing the full children nodes from the deltas and the current node being expanded. We discuss more elaborated restoration strategy of our framework in Chapter 10.

Definition 1.47 (Copying restoration strategy)**.** *For all $q, B \in Q_4$ and $x \in L_3$, we have*

$$\begin{aligned} &\mathit{copy} : Q_4 \times L_3 \times \mathit{Store}(L_3) \to Q_4 \times \mathit{Store}(L_3) \\ &\mathit{copy}(q, x, b) \mapsto (q, \{(l, x \sqcup_3 y) \mid (l, y) \in b\}) \end{aligned}$$

To conclude this part, the branching strategy is often the main component of a search strategy that can be customized in constraint solvers. As such, there is a large collection of branching strategies available in constraint solvers (e.g. GeCode [STL14] and Choco [PFL15]) as well as in modelling languages such as MiniZinc [STF+10].

1.5.4 Exploration strategy

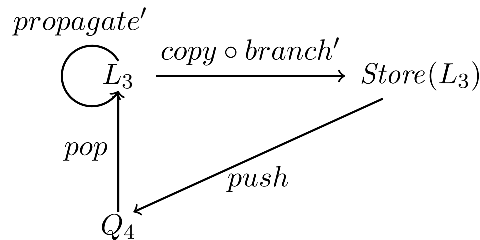


Figure 1.9: Exploration strategy.

We defined several structures isolating different aspects of a search algorithm based on backtracking. In this section, we combine these components together and consider some properties of the exploration strategy obtained. In Figure 1.9, we instantiate the abstract view given in Figure 1.8 with the sub-strategies described so far. This example of exploration strategy is given by the function *explore* assembling the sub-strategies introduced along this section:

$$\begin{aligned} &\mathit{explore} : Q_4 \to Q_4 \\ &\mathit{explore}(q) \mapsto (\mathit{push} \circ \mathit{copy} \circ \mathit{branch}' \circ \mathit{propagate}' \circ \mathit{pop})(q) \end{aligned}$$

We adapt the propagation function similarly to the branching strategy:

$$\begin{aligned} &\mathit{propagate}' : Q_4 \times L_3 \to Q_4 \times L_3 \\ &\mathit{propagate}'(q, x) \mapsto (q, \mathit{propagate}(x)) \end{aligned}$$

Interestingly, the exhaustiveness and compactness properties of the function branch' can be extended without hurdles to the exploration strategy. Indeed,

propagate' and *copy* are order-preserving and do not remove any solution from the problem. Note that even if *explore* is not exhaustive, we can sometimes recover its exhaustiveness using an additional exploration layer (for example with IDS in Example 1.10). However, we leave the treatment of such properties to future works.

1.6 Inference in L_4 with constraint optimization problem

Constraint optimization problems aims at finding the best solution according to some criterion (Section 1.1.2). It is usually solved using the branch and bound algorithm which is a dynamic inference strategy over the lattice L_4. Compared to the inference in L_3, pruning sub-trees not leading to a solution, branch and bound prunes sub-trees not leading to a better solution.

Definition 1.48 (Objective function)**.** *An objective function $f : L_3 \to L_3$ maps an element in Sol_3 to a CSP describing a better solution.*

Example 1.23 (Minimizing objective)**.** A frequent objective function is to minimize the value of a variable at location x:

$$minimize(\langle d, P\rangle) \mapsto \langle \bot, [\![x < lb(d(x))]\!]\rangle$$

We create a constraint $x < lb(d(x))$ specifying that the value of the variable x must be lower than its smallest current value. This constraint is created dynamically according to the CSP $\langle d, P\rangle$. ⌟

Using the objective function, we can define the branch and bound function that will be plugged into our exploration strategy.

Definition 1.49 (Branch and bound)**.** *Let L_3 and L_4 be hierarchical lattices and $f : L_3 \to L_3$ an objective function. Then the branch and bound algorithm is given by the following function:*

$$\begin{aligned}
&bab : Q_4 \times L_3 \to Q_4 \times L_3 \\
&bab(q, x) \mapsto \begin{cases} (\{(l, v \sqcup_3 f(x)) \mid (l, v) \in q\}, x) & \textit{if } ssd(x) = true \\ (q, x) & \textit{otherwise} \end{cases}
\end{aligned}$$

Every time we reach a solution, we constrain all the remaining nodes in the queue with the constraint given by the objective function. It has the effect of forcing the remaining solutions, if any, to be better than the current one.

Proposition 1.10. *The branch and bound function bab is monotonic.*

Proof. By Definition 1.49, we only add information into the queue by increasing the information in all its nodes with the join $\sqcup_3$ which is a monotonic operation. □

We can integrate this function into the basic exploration strategy as follows:

$$push \circ copy \circ branch' \circ bab \circ propagate' \circ pop$$

bab must occur between *branch*′ and *propagate*′ since *branch*′ removes the nodes in the solution space and *propagate*′ creates solution.

A direct implementation of this definition might be problematic when the queue is not fully copied (for example if we use a restoration strategy) because the nodes would not be directly accessible anymore. Instead, we can store the constraint created by the objective function and add it into the future extracted nodes. However, we must manipulate a lattice $Q_4 \times L_3$ where L_3 represents the objective store which requires to adapt the sub-strategies of the exploration function. Therefore, we tackle this variation of branch and bound with the spacetime language later in this dissertation (Section 3.4.2).

1.7 Pruning in L_4 with backjumping

Backtracking as presented in Section 1.5 is more specifically called *chronological backtracking* because it always backtracks to the latest decision made on a variable. However, the reason we fail in a leaf node is not always due to the latest decision. An improvement consists in backtracking earlier in the search tree at the decision responsible for the failure. This optimization is called *backjumping*. It is an example of a pruning strategy over the lattice L_4.

Example 1.24 (Backjumping in a boolean formula)**.** Consider the Figure 1.10 describing the search tree of the following formula

$$(\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge x_3$$

After instantiating x_1 and x_2 to *true*, we try to instantiate the variable x_3 but none of its values is satisfiable with the current partial assignment $\{x_1 \mapsto true, x_2 \mapsto true\}$. In chronological backtracking, we would backtrack to x_2, assign $false$ to x_2 and fail again on x_3. Intuitively, we understand that failure in x_3 is not caused by its direct predecessor but by a decision made higher in the search tree, here the one made on x_1. ⌟

There exists several forms of backjumping [DF02] and, for demonstration purposes, we focus on Gaschnig's backjumping. An important question between backjumping's variants is how they find the point in the search tree responsible for the current failure. In Gaschnig's backjumping, it performs the following steps:

1. For each branch b, find the smallest partial assignment $\{x_1 \mapsto v_1, ..., x_i \mapsto v_i\}$ that is inconsistent with b. If a branch b is consistent with the current assignment, we continue the exploration.

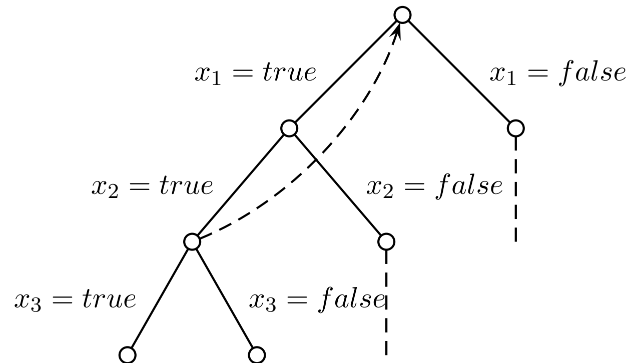


Figure 1.10: Backjumping from x_3 to x_1 with the formula $(\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge x_3$.

2. From all these partial assignments, take the largest one. It corresponds to the highest safe point in the search tree we can backjump to.

If every branch is unsatisfiable, the backjumping point always exists and is unique. It always exists because the branches must be unsatisfiable, at least, because of the latest choice made, in which case we backtrack to the parent's node. It is unique because the current path is a chain of choices, and thus they must be a smallest inconsistent partial assignment (since a chain is totally ordered).

Example 1.25 (Gaschnig's backjumping)**.** On the variable x_3, we have the branches $x_3 = true$ and $x_3 = false$, and their respective smallest inconsistent partial assignments are:

- $\{x_1 \mapsto true\}$ conflicting with $x_3 = true$ due to the clause $\neg x_1 \vee \neg x_3$ being insatisfiable with this assignment.
- The empty partial assignment—the root problem—conflicting with $x_3 = false$ since we have the unit literal x_3 that must be set to true.

From these two partial assignments, $\{x_1 \mapsto true\}$ contains the latest decision variable x_1 and so we jump to this position. With $x_1 = false$, a solution can be found and a part of the search tree has not be explored thanks to backjumping. ⌟

Inference in L_3 is often referred to as a *look-ahead scheme* since it removes values from future instantiations. In comparison, backjumping is a *look-back technique* since it computes over decisions already made. This explains why backjumping is a pruning strategy over L_4: it needs the past decisions contained in the search tree and not only into the current problem. Besides, it does not add information in the search tree; this distinguishes pruning strategies from inference methods. We propose a formal explanation of Gaschnig's backjumping using our lattice framework.

Our search tree is only represented by the current frontier being explored. However, backjumping requires the path of decisions from the root to the current

failed node in order to jump back in the tree. We introduce several convenient functions to manipulate the lattice L_4 as a tree.

Definition 1.50 (Tree operations on L_4)**.** *We define three operations over L_4 for retrieving the root of the search tree, the least common ancestor (LCA) of two nodes and the set of ancestors of a node.*

- *The root of the search tree is the greatest lower bound of the nodes in the queue:*

$$\begin{aligned} &root : L_4 \to L_3 \\ &root(q) \mapsto \textstyle\bigsqcap^3_{x \in q} x \end{aligned}$$

- *Given two nodes $x, y \in L_3$, the greatest lower bound of x and y is also the least common ancestor:*

$$\begin{aligned} &lca : L_3 \times L_3 \to L_3 \\ &lca(x, y) \mapsto x \sqcap_3 y \end{aligned}$$

- *The set of ancestors of a node $x \in q$ for a queue $q \in L_4$ is a chain in L_3 such that it contains all the least common ancestors between x and the other nodes in q:*

$$\begin{aligned} &ancestors : L_4 \times L_3 \to \mathcal{P}(L_3) \\ &ancestors(q, x) \mapsto \textstyle\bigsqcup^{\mathcal{P}(L_3)}_{y \in q} \{lca(x, y)\} \end{aligned}$$

Given the ancestors' set of a node, we can define Gaschnig's backjumping as a pruning function over Q_4. We follow the informal definition given above. Firstly, we define a function hbj computing the highest backjumping node of a branch in the tree:

$$\begin{aligned} &hbj : L_4 \times L_3 \times L_3 \to L_3 \\ &hbj(q, x, b) \mapsto \textstyle\bigsqcap \{a \in ancestors(q, x) \mid ssd(a \sqcup b) = false\} \end{aligned}$$

Admitting we are in a node x and considering a branch b, the highest backjumping node of x is its highest parent that is failed with b. Since the ancestors' set forms a chain in L_3, its greatest lower bound is the minimal element of the chain and thus the highest parent. If such an element does not exist, then $hbj(q, x, b) = \top$ since $\top$ is the identity element of $\sqcap$.

Secondly, given a set of branches, we compute a safe backjumping node:

$$\begin{aligned} &sbj : L_4 \times L_3 \times Store(L_3) \to L_3 \\ &sbj(q, x, B) \mapsto \textstyle\bigsqcup^3 \{hbj(q, x, b) \mid b \in B\} \end{aligned}$$

For every branch, we compute its highest backjumping point. This time, we want the lowest node in this set since we do not want to backtrack too high in the search tree and miss a solution. Therefore, sbj maps to the least upper bound over the

set of the highest backjumping ancestor of each branch. If a branch can actually lead to a solution, then sbj maps to $\top$ since $\top$ is the absorbing element of $\sqcup$.

Finally, we define Gaschnig's backjumping by removing all the nodes in the queue below the highest safe backjumping node.

Definition 1.51 (Gaschnig's backjumping)**.** *We store the highest safe backjumping node in bj. Whenever $bj = \top$, it means that we cannot backtrack yet and gaschnig acts as the identity function. In the other case, we remove the nodes below or equal to bj from the queue.*

$$\begin{aligned}
&\mathit{gaschnig} : Q_4 \times L_3 \times \mathit{Store}(L_3) \to Q_4 \times L_3 \times \mathit{Store}(L_3) \\
&\mathit{gaschnig}(q, x, B) \mapsto \begin{cases} (q, x, B) \;\; \mathit{if}\; bj = \top \\ (\{(l, v) \in q \mid bj \neq (v \sqcap bj)\}, x, \bot) \;\; \mathit{otherwise} \end{cases} \\
&\quad \mathit{where}\; bj = sbj(\pi'_2(q), x, B)
\end{aligned}$$

The expression $bj \neq (v \sqcap bj)$ filters out the children of bj in the queue.

Backjumping can be integrated into the full exploration strategy using a branching strategy enumerating one variable at each level. Finally, the exploration strategy based on Gaschnig's backjumping is assembled as follows:

$$\mathit{push} \circ \mathit{copy} \circ \mathit{gaschnig} \circ \mathit{enumerate} \circ \mathit{propagate}' \circ \mathit{pop}$$

It is integrated seamlessly in the existing exploration strategy and could be combined with branch and bound as well.

1.8 Conclusion and discussion

The main contribution of this chapter is to lay down a novel hierarchical framework for constraint programming based on lattice theory. We show that seemingly disparate techniques are defined over the same underlying combinatorial structures. We now discuss several opportunities fostered by this theory and give some limitations of this framework. Firstly, this formalism encompasses most of the components of a constraint solver. Therefore, it should be possible to use an automatic theorem prover to generate a verified constraint solver. Furthermore, a number of properties including exhaustiveness, termination and compactness have been very little explored here. A challenging aspect would be to automatically verify these properties on user-defined search strategies. Secondly, the propagation needs a more in-depth treatment, following the presentation in [Tac09], especially considering event-based programming. Thirdly, this chapter does not do justice to many important algorithmic techniques including no-goods learning [DF02], lazy clause generation [OSC09] and local search [VHM05].

No-goods are constraints inferred from L_4 when reaching a fail state in L_3. They are useful to avoid repeatedly exploring similar subtrees that would fail for an identical reason. It can be used in association with backjumping since this

last actually computes a no-good when returning to a higher node. Compared to propagation, no-goods learning can be viewed as an inference method over L_4 instead of L_3.

Lazy clause generation relies on the efficiency of SAT solvers for solving CSPs. To achieve that, it generates on-the-fly SAT clauses encoding propagators as the search progresses. The interplay between SAT solving and traditional propagation has been shown to drastically improve the performance on many finite domains problems.

Local search is a non-exhaustive search technique successful for finding solutions to large problem instances for which exhaustive search—or in contrast "global search"—would take too much time. It directly works on ground values and not on domains, and thus L_0 is equal to L_1. Various algorithms exist but most are captured by the following idea as introduced in [VHM05]. Given a state $S \in L_3$, we move to a neighbouring state such that this state is closer to a solution. Moving from one state to the other is not necessarily a monotonic operation over L_3—a reason why local search is not exhaustive. Since the variables are ground, the states manipulated are in $Fail_3$ and it tries to come closer to Sol_3 according to a neighbourhood distance. For example, we move to a state such that the number of violated constraints decreases—this is a monotonic function but we do not have the guarantee we can reach Sol_3.

To conclude this chapter, we hope that digging further into that direction might lead to an unified theory of constraint programming. A striking example of such unification is the delta operator Δ that is fundamental in many techniques including restoration strategy, backjumping but also local search when measuring the impact of a move compared to another. In the next chapter, we survey constraint-based programming languages with the lattice hierarchy developed in this chapter.

Chapter 2

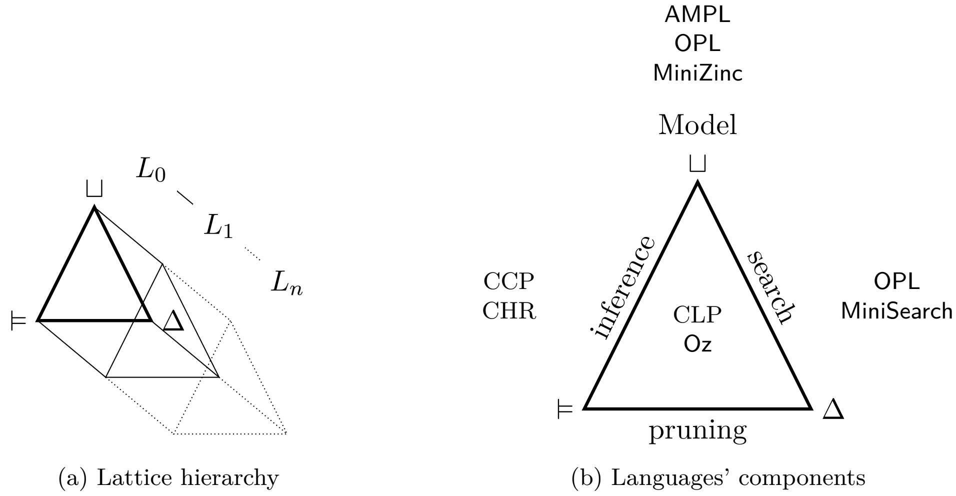
Constraint-based Programming Languages

2.1 Introduction

A constraint-based programming language is a language equipped with variables defined at least over L_0 and with relations over these variables by proposing the operator $\sqcup_3$. Since its inception, constraint programming is actively combined with various paradigms ranging from logic to imperative and functional programming. The logic paradigm is one of the most successful and frequently studied combinations between algorithmic and language aspects of constraint programming. However, the past two decades witnessed an increasing interest in constraint libraries due to their better efficiency and integration into mainstream languages. These are the main reasons why the literature on constraint programming is cleaved in solving algorithms and language abstractions. The goal of this chapter is to survey constraint-based languages with the lattice hierarchy introduced in the first chapter. We also draw various links between algorithmic approaches and their relevant abstractions in programming languages. As we portray in Figure 2.1a, the hierarchical dimension of our framework multiplies the possible operators inside a language. Therefore, considering all the combinations of operators, it is natural that a large panel of languages exists.

Nowadays, the algorithmic aspect of constraint programming is predominant as attested by numerous solvers packaged as libraries. Many constraint problems that occur in practice are intractable—there is no existing algorithm to solve them in polynomial time. Nevertheless, when analysing a problem, we can often customize the constraint solver to explore the state space of the model more efficiently. For example, QuickPup [TFF12] is a search strategy tailored to solve real-world instances of a configuration problem. In this regard, a recurring problem is the difficulty to tweak constraint libraries whereas language abstractions can drastically simplify the tweaking process. This fact is corroborated by recent works devoted to abstractions for specifying the search strategy in a constraint solver [STW$^+$13, SDTD14, MFS15].

However, designing new constraint-based languages is a daunting task, partly due to the long history of the field which began with logic programming, and



(a) Lattice hierarchy (b) Languages' components

Figure 2.1: Algebraic structure of a constraint-based language.

partly because of the scarce work that studies and compares the existing languages with constraints as a central notion[1]. In this respect, we find many constraint-based programming languages that extend various paradigms and use distinct algorithmic facets of constraint programming. Therefore, it is hard to understand what language is suitable for one task, and how this language is relevant to solving algorithms. Ideally, a constraint-based programming language should be based on the efficiency of modern constraint solving, and provide useful abstractions to customize and program these solving algorithms.

In Figure 2.1b, we relate the algebraic operators of the lattice framework and the three main ingredients of constraint programming: model, inference and search. A constraint-based language is usually specialized to one component although languages attempt to merge two or the three of these components. We do not include pruning into our classification because we do not know of any language centered around pruning. However, we show along this survey that some languages from all categories include pruning abstractions, for example the *cut* in Prolog.

Model

Modelling languages have deep roots into first-order logic—reviewed in Section 2.2.1—since it is usually a high-level abstraction to specify logic formulas. Examples of such modelling languages include AMPL [FGK90], OPL [VHMPR99] and MiniZinc [NSB+07]. These languages provide modelling abstractions and delegate the solving part to specialized constraint solvers. We illustrate this paradigm through several constraint models in Section 2.3.

[1] Although several works exist in the more general context of multi-paradigm languages (e.g. [RH04, Hof11, Han14]).

Inference

Language theoretic research on constraint inference stems from the field of constraint and concurrent logic programming [Sha89]. Examples include concurrent constraint programming (CCP) [SR89] and constraint rewriting systems such as constraint handling rules (CHR) [Frü98]. This class of language is often associated with the notion of "don't care nondeterminism" which models nondeterminism without backtracking. One of the common limitations of such languages is to consider convex constraint systems—those that do not need a search step to be solved. However, this restriction allows backtrack-free languages to define concurrent and infinite computations. We give an account of theses languages in Section 2.4.

Search

For the efficiency reasons mentioned earlier, it is essential for a language to provide support for expressing search strategies. On the one hand, a large piece of work has been dedicated to bridging inference and search inside a unique language. This is notably the case of constraint logic programming (CLP) and we study it in depth in Section 2.5. On the other hand, due to some difficulties to merge L_3 and L_4, various search-only languages—usually accompanying a modelling language—have been developed. For example, a model written in MiniZinc can be conjointly used with the MiniSearch language [RGST15]. We review languages supporting search as well as their limitations in Section 2.6.

In sum, we provide a comprehensive survey of the main constraint-based programming languages using the lattice framework of Chapter 1. It illustrates that algebraic operators provide the necessary connections between the algorithmic and language aspects of constraint programming. We demonstrate that lattices are expressive enough to formally explain a large variety of constraint-based programming languages. Specifically, we reinterpret the semantics of CCP and CLP with the algebraic operators $(\sqcup, \models, \Delta)$.

We organise this chapter by classifying languages according to their levels in the lattice hierarchy. In order to clearly understand the survey starting in Section 2.3, we first provide the necessary background in logic and a few historical notes in the field of logic languages.

2.2 Background

In order to clearly understand the links between language and algorithmic approaches to constraint programming, it is useful to take a look at their evolution chronologically. Therefore, we start with a short account of first-order logic and the computational models that follow this mathematical theory.

2.2.1 First-order logic

The main goal of programming languages is to formalize a set of concepts that interact well in order to solve a substantial class of problems. Actually, Hilbert—through the so-called Hilbert's program—already had a similar objective albeit more ambitious: finding a set of axioms that can formalize all the mathematics. This problem culminated in the modern syntax of first-order logic (FOL) given by Hilbert and Ackerman [HA28] in 1928. We give some terminologies surrounding the syntax of FOL (based on [Seb07]) used throughout this chapter.

Definition 2.1 (Syntax of first-order logic)**.** *A first-order signature S is a tuple (V, F, P) where V is the set of variables, F the set of function symbols and P the set of relation symbols—called predicates. We note $x, y, \ldots \in V$ the variables, $f, g, \ldots \in F$ the function symbols and $p, q, \ldots \in P$ the predicate symbols. Each function and predicate symbol has an arity $n \geq 0$. A function symbol (resp. predicate symbol) with an arity of 0 is called a constant (resp. boolean atom). A term is either a variable or a function whose arguments are terms. An atom is a predicate whose arguments are terms. A literal is either an atom a or its negation $\neg a$. A formula is built by the usual existential quantifier $\exists$, the universal quantifier $\forall$ and the (non-minimal) set of boolean connectives $\{\neg, \wedge, \vee, \longrightarrow, \leftrightarrow\}$. We say that a formula ϕ (or term t) is closed (or ground) if there is no variable free in ϕ (or t). A sentence is a closed formula. A theory is a set of sentences with a signature S. A clause is a disjunction of literals $l_1 \vee \ldots \vee l_n$. A formula in conjunctive normal form (CNF) is a conjunction of clauses.*

We consider a model-theoretic semantics, pioneered by Tarski in 1933 [Tar33], which is the prominent semantics adopted in the field of constraint-based languages.

Definition 2.2 (Model-theoretic semantics of first-order logic)**.** *Given a signature $S = (V, F, P)$, an interpretation[2] I is a tuple (D, F_I, P_I) where (i) D is a non-empty set of elements—called the domain of discourse, (ii) F_I is a function mapping function symbols $f \in F$ with arity n to interpreted functions $f_I : D^n \rightarrow D$, and (iii) P_I is a function mapping predicate symbols $p \in P$ with arity n to interpreted predicates $p_I \subseteq D^n$. Given a closed formula ϕ built from S, an interpretation I is said to be a model of ϕ if I is true in ϕ. Equivalently, we say that I satisfies ϕ.*

In addition to the syntax, Hilbert and Ackerman raised three fundamental questions:

1. "Can every valid sentence in FOL be proven to be valid within FOL?" (completeness),

2. "Does a formal system never permit to derive a sentence and its negation?" (consistency), and

[2] Also called a structure in the model theory.

3. "Is there a decision procedure proving the validity of every sentence in FOL?" (decidability).

Soon after, Gödel proved that FOL is complete [Göd30]. However, by its first incompleteness theorem [Göd31] of 1931, he showed that it is impossible to provide a formal system—at least capable of encoding elementary arithmetic—that is both complete and consistent. Hence FOL is an inconsistent formal system in which some sentences cannot be proven true nor false. The existence of a generic decision procedure for FOL was later rejected by the Church-Turing thesis [Chu36, Tur37]. There are two approaches in programming languages to overcome the undecidability result of FOL: either accepting that sometimes the algorithm will not terminate or considering a decidable fragment of FOL.

2.2.2 Semi-decidability

Accepting that an algorithm may not terminate is historically the first computational approach to tackle the undecidability of FOL. Logic programming is one of the most striking examples of trading decidability for expressiveness[3]. In practice, the burden of undecidability is delegated to the programmer who is responsible for limiting in time the computation, and to cope with runtime errors such as stack overflow. We first give some historical remarks about computational logic, and then we consider logic programming.

One of the central results is Herbrand's theorem [Her30] showing that FOL is actually semi-decidable, that is, for any valid FOL sentence F, there is a proof of finite length. This theorem is of paramount importance to distinguish the foundation for constraint solving (satisfiability) and theorem proving (validity). To understand it, we define the notions of Herbrand universe and Herbrand base followed by an algorithm to decide the validity of a formula.

Definition 2.3 (Herbrand universe)**.** *Given a first-order signature* (V, F, P) *with* F *containing at least one constant symbol, its Herbrand universe* H_u *is the set of all ground terms that can be built out of* F*.*

Definition 2.4 (Herbrand base)**.** *The Herbrand base* H_b *is the set of all ground formulas that can be built out of* H_u*.*

In 1960, Davis and Putnam [DP60] introduced a decision procedure for FOL that performed better than the contemporary approaches by avoiding their (almost) systematic exponential blow-up. It interleaves the following two steps (the formula is first preprocessed to be in CNF):

(i) it instantiates the variables with terms of the Herbrand universe, and

(ii) it proves (or disproves) the validity of the obtained quantifier-free formula.

[3] In this case, the gain in expressiveness is the Turing completeness of the language.

Given a formula F in CNF and its Herbrand universe H_F, the formula can be instantiated into $|H_F|$ subformulas forming a new (possibly infinite) CNF formula $F_1 \wedge F_2 \ldots \wedge F_n$. This is called the grounding step because it "grounds" to values the variables of F. If a subset of formulas $F_1 \wedge \ldots \wedge F_i$ is proven unsatisfiable, then the whole formula is unsatisfiable. Dually, if it is proven valid, we need to continue on the extended conjunction of formulas $F_1 \wedge \ldots \wedge F_{i+1}$. The Herbrand's theorem tells us that if a formula is unsatisfiable, then there is a finite conjunction of formulas that is unsatisfiable. Note that the validity is proved with the negated formula $\neg F$. Indeed, if a formula is valid, its negation is unsatisfiable. Answer set programming (ASP) is a paradigm for solving FOL formulas which is based on a grounding step, and thus ASP has very efficient grounders [Bar03].

The second step, especially in the refined and implemented version of 1962 [DLL62], called the DPLL algorithm from the names of its authors, is still at the heart of modern SAT solvers. Its main inference rule is:

$$\text{split}\frac{c \vee l \quad d \vee \neg l \quad r}{(c \wedge r) \vee (d \wedge r)}$$

This rule is better understood top-down since it generates a search space: $c \wedge r$ assumes that l is valid, and $d \wedge r$ that l is invalid. A contradiction is obtained whenever this rule generates the empty clause—it is obtained only from $l \wedge \neg l$ which is always false. If $c \wedge r$ is proven invalid, then we can try to prove the validity with $d \wedge r$. On the contrary, if $c \wedge r$ is proven valid, then the whole formula is valid. In fact, this splitting rule is central to any solving algorithm based on backtracking. In addition, DPLL comes with unit propagation and pure literal elimination for, respectively, propagating clauses with only one literal (which must be true) and removing literals that only appear positively or negatively in the formula.

Generating all the grounded formulas is not always practical and efficient. Hence, a generalization of this rule for quantified formulas, called the resolution principle, has been developed by Robinson in 1965 [Rob65]. It is based on earlier work of Prawitz who rediscovered unification[4] and proposed his own version of a proof-finding algorithm [Pra60]. An elegant fact is that resolution is a complete inference system for the first-order logic constituted of a single rule. The resolution principle is at the heart of the logic programming paradigm that we introduce next.

2.2.3 Logic programming

Early in the seventies, researchers tried to puzzle out how to use logic in an efficient way. At that time, the debate was between procedural (e.g. Planner [Hew69, Hew71]) and purely logical (e.g. QA3 [Gre69]) representation of knowledge. This debate lasted until the team of Colmerauer—developing a natural

[4]Actually, David and Putnam proof-finding procedure as well as unification were already shown in the thesis of Herbrand respectively under the names of Property B and Property A [Rob00].

language question-answer system—and Kowalski noticed that a particular form of logic formula, named Horn clauses, could be interpreted logically and procedurally [Kow74]. These findings gave birth to the well-known Prolog programming language. In the following, we consider a modern version of Prolog and we do not review the many variants of the resolution rule in the context of Prolog. Additional references can be found in [Col93] for Prolog, [Hew09] for the procedural vision (e.g. Planner) and [Rob00, Kow14] for a more general review of the logic programming history. Note that combining declarative (logic) paradigms and efficient (procedural) solving algorithms is still a major challenge today, as we will see in the forthcoming sections. We now give some terminologies relevant to Horn clauses.

Definition 2.5 (Horn clause)**.** *Horn clauses are clauses with exactly one positive literal. They are represented by the formula* $A_0 \vee \neg A_1 \vee \ldots \vee \neg A_n$ *(*$n \geq 0$*) where* A_i *is an atom, and more commonly by the implicative formula* $A_0 \leftarrow A_1 \wedge \ldots \wedge A_n$*. The atom* A_0 *is called the head and the conjunction* $A_1 \wedge \ldots \wedge A_n$ *the body. In case* $n = 0$*, then* $A_0 \leftarrow$ *true is called a fact and is simply noted* A_0*. In case* $A_0 = false$*, then* $false \leftarrow A_1 \wedge \ldots \wedge A_n$ *is called a goal and is noted* $\leftarrow A_1 \wedge \ldots \wedge A_n$*.*

Definition 2.6 (Logic program)**.** *A logic program is a set of Horn clauses.*

Definition 2.7 (Normal logic program)**.** *An important extension of Horn clauses, named the general clause, is to accept negative literal in the body. A normal logic program is a set of general clauses.*

Given a logic program P and a goal G—representing the question to ask—we can give a logic semantics to Prolog. By negating the goal G, we obtain the formula $P \wedge \neg G$ in CNF that can be sent to a theorem prover, using an algorithm such as the one described above. If the formula is valid, then the query G is unsatisfiable, otherwise we obtain a contradiction which is an answer to the problem. Since Prolog focusses on Horn clauses, the resolution principle has been adapted and improved for this particular case; the algorithm is named SLD-resolution [Kow74]. In case of normal logic programs, a negated literal is considered $false$ if we cannot prove it—this principle is called the negation as failure (NAF). The most traditional semantics for NAF is the Clark's completion [Cla78] where a program is considered closed: we know everything so a literal that we cannot prove is considered false. This is a form of non-monotonic reasoning because if we add new facts afterwards, a literal considered false might become true. Additional insights on the different resolution rules, NAF and the relation of Prolog to logic can be found in [NM95]. A standard book on Prolog is [SS94], and Prolog in the context of artificial intelligence is explained in [Bra12]. We consider in more details a generalized version of the semantics of Prolog in the context of constraint logic programming in Section 2.5.2.

2.2.4 Decidable fragments of FOL

Restricting FOL to some decidable fragments makes it amenable to fully automatic algorithms. There are a large amount of interesting fragments and various communities propose specific approaches to solve some fragments. In the following, we briefly introduce the main four approaches: SAT, constraint programming, satisfiability modulo theories (SMT), and (mixed) integer programming.

The restriction of FOL to propositional logic has been explored in depth in the SAT community [BHvMW09]. A solver SAT takes a formula in CNF such as $(a \vee b) \wedge (\neg b \vee c)$ and answers with a truth model of this formula, for example $\{a \mapsto true, b \mapsto false, c \mapsto false\}$. The domain of discourse of a solver SAT is the boolean set $\{true, false\}$. We can encode problems modelled with integer values into propositional formulas—which is feasible since in computer science, everything is encoded with bits anyway. However, encoding any problem into SAT can results in million of clauses for a single constraint. Another problem is that we lose the structure of the initial problem which can bring valuable information to solve the problem faster. The following approaches use logic predicates to preserve the overall structure of the problem.

Constraint programming mainly relies on FOL formulas without quantifiers and over finite domains [RvBW06]. We studied this approach in Chapter 1.3. Intuitively, this fragment of FOL is decidable since, to establish the satisfiability of any formula, it is sufficient to enumerate all the models. In short, a constraint solver takes a formula such as $x, y, z \in \{1, 2, 3\} \;\wedge\; x + 1 < y \;\wedge\; x \neq z$ and finds a model satisfying the formula such as $\{x \mapsto 1, y \mapsto 3, z \mapsto 2\}$. Similarly to SAT, a constraint problem is generally NP-complete; the challenge is to find an efficient algorithm for this generic settings—compared to SAT which has more specialized and efficient algorithms for boolean domains. In this context, hybrid approaches are explored by lazily generating the SAT formula and keeping the overall structure of the problem [OSC09]. Historically, this paradigm has evolved together with programming languages, and so is central to this survey.

Many fragments of FOL, called theories, are extensively studied in SMT solvers (see e.g. [Seb07]). In contrast to general-purpose solving algorithms, theories help to design efficient and specialized algorithms, and to study the properties of these theories such as their algorithmic complexity. The complexity of a theory ranges from polynomial (e.g. linear arithmetic over reals) to undecidable for the full FOL. The goal is to find theories that offer an interesting trade-off between expressiveness (the kind of problems it can model) and complexity. We define two theories that will be relevant to the programming languages studied later. Similarly to [Seb07], all theories contain the equality predicate $=$ which is a reflexive, symmetric and transitive relation, and a congruence.

Definition 2.8 (Theory of equality and uninterpreted functions)**.** *The quantifier free theory of equality and uninterpreted functions (EUF) has any signature* $S = \langle V, F, P\rangle$ *and does not interpret the function and predicate symbols—hence the*

terminology "uninterpreted". More exactly, the semantics interpretation functions are the identity functions. Therefore, the domain of discourse is the Herbrand universe of the signature S.

The theory EUF is central to logic programming (Section 2.2.3).

Definition 2.9 (Theory of linear arithmetic)**.** *The quantifier free theory of linear arithmetic over integers ($LA(\mathbb{Z})$) has the signature $\langle V, F = \{+, *\} \cup \mathbb{Z}, P = \{<, >, \neq, =, \leq, \geq\}\rangle$ with atoms of the form $c_1 * x_1 + \ldots + c_n * x_n \,\Box\, c_0$ where $\Box \in P$, c_i is an interpreted constant and $x_i \in V$ a variable. The semantics consist of the domain of discourse $\mathbb{Z}$ and the symbols are interpreted as in standard arithmetic. We refer to $LA(\mathbb{Q})$ when the theory ranges over the rationals and $LA(\mathbb{R})$ when it ranges over the reals.*

The SMT solvers work together with a SAT solver in two steps: (i) the atom of the formula are abstracted, for example, $x > y \vee x \neq y$ is turned into $a \vee b$, and the latter formula is sent to a SAT solver, and (ii) the SAT solver returns a model, and the corresponding atoms (or the negation of these atoms) are solved within their respective solver. In case of non-satisfiability, a new SAT model is queried. Of course, modern SMT solvers take a lazy approach in the sense that these two steps are intertwined [Seb07]. We call this algorithm $DPLL(T)$ where $DPLL$ is the algorithm seen above to solve propositional formulas and T is a theory that is instantiated in this framework. The formal and practical study of the combination of two or more theories is one of the challenges in a SMT solver. More information on the solving algorithms behind these theories as well as techniques to combine these theories is available in e.g. [KS08, Seb07].

Finally, the approach of linear programming is dedicated to the theory of linear arithmetic. It comes in various flavours depending on the domain of discourse: linear programming over real numbers, integers (integer linear programming), or both (mixed integer programming). Linear programming algorithms are tailored for constraint optimization problems which aim at finding the best solution according to some criteria.

It is important to keep in mind that these approaches are not exclusive, and many works attempt to combine their advantages [Hoo02, Hoo12, VH02, OSC09, BG12]. In the following, we mainly focus on constraint programming as it shapes most of the constraint-based programming languages.

2.3 Modelling languages: from L_0 to L_4 ($\sqcup$)

2.3.1 Declarative languages

The pioneering ALICE language

ALICE [Lau78], designed around 1976 by Jean-Louis Laurière, is a pioneering constraint modelling language. Surprisingly, the structure of an ALICE program shares

much in common with modern modelling languages. Despite its lack of support for abstractions, ALICE's primitives can already express a large number of problems. The underlying combinatorial objects are sets and functions: we model in terms of set correspondences. For example, it supports the modelling of injective, surjective and bijective functions between two sets—essential to model assignment problems. To demonstrate this, we consider a scheduling problem and its ALICE model as appearing in [Lau78]. This will serve as a starting point to explain two modern modelling languages: AMPL [FGK90] and MiniZinc [NSB+07], respectively from the mathematical and constraint programming communities.

Example 2.1 (Congress scheduling)**.** The organizers of a congress have filled the participants' preferences in a matrix M where $M_{i,j} = 1$ if there is a congressman who wants to attend both the sessions i and j. There are 11 sessions dispatched in 3 rooms. The session 10 comes after the session 5, and the session 11 after the fourth and sixth. Each session lasts for one time slot. Find a schedule such that all the constraints are satisfied and the duration of the congress is minimized. ⌟

An ALICE model captures the problem as follows:

```
GIVEN CST N
      SET S = INT 1 N        (sessions)
          H = INT 1 N        (periods)
FIND  FUN F ↣ S  H    DIS        DMA
WITH  MIN MAX F I   (disjunction) (maximum degree)
      F 5 < F 10
      ( F 11 > F 4 ) AND ( F 11 > F 6 )
END
```

The model is clearly separated into constants (N), variables (S and H), constraints and an optimization function. The goal is to find a map F between the set of sessions S and the set of periods H. The expression `MIN MAX F I` is an objective function that minimizes the maximum element of the function F. Several simple constraints follow, e.g. `F 5 < F 10`. The core of the problem resides in the constraints DIS and DMA put on the map F. The constraint DIS ensures that two distinct elements x, y in the domain of F map to a distinct element ($F(x) \neq F(y)$); the list of such distinct elements must be given as an input. The constraint DMA ensures that each element y in the image of F is mapped onto at most n times by every element in the domain of F. Here, the input of DIS is the preference matrix and the one of DMA is a vector $(3, 3, \ldots, 3)$ since only 3 rooms are available at any time.

Algebraic modelling languages

ALICE is a language bundled with its own solver; at that time, there were not so many different solving algorithms and generic schemes were not yet well-understood. Besides ALICE, research in modelling languages was fostered by industrial needs to program more easily mathematical models. These include optimization problems

over real numbers—(non-)linear programming—and over real and integers, namely mixed integer (non-)linear programming. In comparison, constraint programming is more general but less efficient on solving these particular problems. Languages targeting this class of problems are usually called algebraic modelling languages (AML). We model the "Congress scheduling" problem using AMPL [FGK90], an early AML language which is still maintained today.

```
param n;
set S ordered = 1..n;
set H ordered = 1..n;

param DMA {H} > 0;
param DIS {S, H} binary;

var F {S} >= 1 <= n integer;
var Y {S,H} binary;

minimize Duration: max {j in S} F[j];

subject to Precedence1: F[5] + 1 <= F[10];
subject to Precedence2: F[11] >= F[4] + 1;
subject to Precedence3: F[11] >= F[6] + 1;

subject to dis1{i in S, j in S: i < j and DIS[i,j]==1}:
  F[i] - F[j] + 1 <= 30 * Y[i,j];
subject to dis2{i in S, j in S: i < j and DIS[i,j]==1}:
  F[j] - F[i] + 1 <= 30 * Y[j,i];
subject to dis3{i in S, j in S: i < j and DIS[i,j]==1}:
  Y[i,j] + Y[j,i] = 1;

subject to dma{k in H}:
  numberof k in ({j in S} F[j]) <= DMA[k];
```

The model is divided into five parts: the user data (indicated by `param`), the constants (here the sets), the variables (e.g. the function `F`), the objective function (`minimize Duration`) and the constraints (starting with `subject to`). It clearly shows the important legacy—or most probably the rediscoveries[5]—left behind by ALICE. The AMPL model is almost organized identically to the ALICE model; the only change is the nature of the `DIS` and `DMA` constraints. Most of the solvers behind AMPL only provide the operators $\{\geq, \leq, =\}$, forbidding strict inequalities. This is an example of trade-off between language expressiveness and efficiency of the underlying solver. The preferences of the congressmen, consisting of a conjunction of constraints $F[i] \neq F[j]$, must be modelled using the operators available only. For this purpose, we use the method of the "big M"—proper to linear programming—where each constraint $\neq$ is broken into three "less or equal than" constraints. The second constraint `DMA` is modelled using a constraint programming extension to

[5]See the preface of Jacques Pitrat in [Lau96a].

AMPL ensuring that the number of sessions per period does not exceed 3. We do not cover the history of algebraic modelling languages here, and we refer the reader to e.g. [KPH04].

Constraint modelling languages

Constraint programming generated many families of programming languages, starting with constraint logic programming (see Section 2.5.2), and later on with constraint modelling languages. Due to the generality of the constraint paradigm, it is very important to use a solving algorithm adapted to the problem (for example a linear solver if the problem can be modelled this way). This motivated solver-agnostic modelling languages: the model is compiled into a simpler intermediate language that can be read by many different solvers. It becomes very straightforward for the user to test a model with different solver and to compare their efficiency on a particular problem. However, creating such a standard is not easy: it must spread among the community and multiple solvers must support it.

We consider MiniZinc [NSB+07], a standard modelling language for constraint problems. A MiniZinc model is compiled to the intermediate language FlatZinc, which removes some of the complexity by unrolling and flattening many expressions which are then read by a solver. There are dozens of solvers supporting the FlatZinc format including constraint solvers but also mixed integer programming, SAT and SMT solvers [SFS+14]. This success is partly due to the MiniZinc competition. It was introduced around the same time as the language for encouraging its use as a *de facto* standard [SBF10]. It gives the opportunities to the developers to compare their solvers in exchange for a small investment for supporting the FlatZinc format.

Another important key point of constraint modelling languages is to provide abstractions and global constraints. For a last time, we consider the "Congress scheduling" problem, but this time in MiniZinc.

```
int: n;
set of int: S = 1..n;
set of int: H = 1..n;
array[int] of int: DMA;
array[int,int] of int: DIS;
array[S] of var H: F;

constraint global_cardinality_low_up_closed(F, [i|i in H], [0|i in S], DMA);
constraint forall(i in S, j in S where i < j /\ DIS[i,j]=1)
                (F[i] != F[j]);
constraint F[5] < F[10] /\ F[11] > F[4] /\ F[11] > F[6];
solve minimize max(F);
```

For explanations on the MiniZinc syntax, please refer to Section 1.1.1. We focus on the global constraint `global_cardinality_low_up_closed(F,I,L,U)` which generalizes the DMA constraint of ALICE. It restricts the values in the array F to take, for an element i, each value in I[i] between L[i] and U[i] times. Global constraints

are a fundamental concept behind the success of constraint solvers. Indeed, they enable modellers to capture combinatorial sub-structures of a problem in a similar way that procedures capture sub-problems in the imperative paradigm. Global constraints have at least three advantages:

(i) they can be re-use across models,

(ii) solvers can provide efficient propagators' implementations, and

(iii) global constraints can be decomposed into primitive constraints, and so are still usable with solvers not directly supporting them.

In the AMPL model, we used the global constraint `numberof` for modelling the restrictions on the number of rooms. Actually, to solve this constraint, AMPL must be used together with a constraint programming extension. An active research field is to merge these different algorithmic approaches into a single language. In this case, if the model is mixed with different constraint kinds, the challenge is to make the different underlying solvers cooperate. As mentioned earlier, a simple but successful cooperation scheme is the one provided in SMT solvers where theories communicate through equality constraints. From a language perspective, OPL [VHMPR99] is a language specifically designed to merge mathematical programming (e.g. mixed linear programming) and constraint solving. In particular, they give several examples in [VH02] demonstrating how integer programming techniques such as cuts—a central concept to reduce the search space in linear programming—can be improved using information from global constraints.

Modelling with $\{\sqcup_0, \sqcup_1, \sqcup_2, \sqcup_3, \sqcup_4\}$

Interestingly, one could argue that ALICE comes closer to a mathematical definition than mainstream modelling languages. We attempt to show that this is mostly a syntax concern. Indeed, all three models have actually similar connections to our lattice hierarchy.

- Constants are members of the lattice L_0 and integer sets are defined over L_1. The operations $\sqcup_0$ and $\sqcup_1$ are usually not language primitives as such. However, these two operations are used when merging the model with its data. In the AMPL model, the instruction `param n` can be seen as n initialized to $\bot_0$: we do not yet have any information about n when looking just at the model. When, in the data file, the instruction `param n := 11;` is reached, it actually performs the operation $\bot_0 \sqcup_0 11$. If n is initialized to two different values, then we reach $\top_0$ which is normally statically forbidden. It works similarly with the integer sets in L_1. In FlatZinc, the compiled model does not contain variables defined over L_0 and L_1 anymore and only makes use of the operations $\sqcup_2$ and $\sqcup_3$.

- In ALICE, the map F is defined over L_2 where the domain of the function is the location set and the image is the value set of L_2. Similarly in MiniZinc, the array `array[S] of var H: F` is defined over L_2 where the indices S form the location set and the cells—of type `var H`—constitute the value set. The statement `array[1..n] of var 1..n` declares n variables which corresponds to the element $\bigsqcup_{l \in [1..n]} \{(l, [1..n])\}$ in L_2.
- Constraints are defined over L_3. Taking MiniZinc as an example, the statement `constraint` c corresponds to the join of the current CSP with c written as $\langle d, P\rangle \sqcup_3 \langle d, \{[\![c]\!]\}\rangle$. The n-ary conjunction `forall(`r`)(`c`)` generates the CSP $\bigsqcup_{i \in r} \langle d, \{[\![c]\!]\}\rangle$ that can be joined into the main CSP.
- The optimization function is joined in L_4 during the solving time (see Section 1.6).

2.3.2 Constraint imperative programming

It is sometimes difficult to integrate a constraint model into a larger non-declarative program. This is especially true when the model is built dynamically at run-time and depends on user interactions. Indeed, pure declarative languages assume that the model is statically known. This motivated the creation of constraint imperative programming[6] languages incorporating constraints into their semantics.

Among the languages in this family, Kaleidoscope [LFBB94], designed around 1991, is one of the first full-fledged oriented-object programming languages integrating constraints. Twenty years later, the ideas developed in Kaleidoscope are re-explored with the language Babelsberg/R [FBH13] extending the programming language Ruby and Babelsberg/JS [FMBH15] extending Javascript. Meanwhile, the language Turtle [Gra03] has explored a simpler integration into the imperative paradigm without support for objects. We discuss these languages from our lattice hierarchy viewpoint.

Compatibility with the imperative paradigm

To stay compatible with the lack of domains in imperative languages, the domains of the variables are not directly accessible in Kaleidoscope. From a modelling perspective, the lack of the join $\sqcup_1$ is not a problem since we can always use constraints to intentionally represent domains, e.g. $\langle d, P\rangle \sqcup_3 \langle \bot_3, \{[\![x \geq 2]\!], [\![x \leq 4]\!]\}\rangle$ instead of $d(x) \sqcup_1 [2..4]$. However, the lack of domains implies that the user only manipulates fully instantiated variables, and thus we cannot observe partial information. This design is coherent with the lack of entailment operators in these languages since it would not be possible to query these partial results anyway.

An important incompatibility with the nature of imperative languages is the unpredictability of the execution time. Time guarantees are often difficult to

[6]This term was coined in the context of the language Kaleidoscope [FBB92] but the concept itself dates back to languages integrating backtracking (Section 2.6.2).

assess when solving a constraint problem and this can be problematic in graphical interfaces, for example. A partial solution in Babelsberg is to build on incremental capabilities of constraint solvers when available in order to improve the solving time.

Dynamic construction of the model

In contrast with purely declarative approach, the constraint imperative paradigm interleaves modelling phases where new constraints are added, and solving phases where the variables are instantiated. The solving phase is usually transparent to the user who cannot control its activity—a reason why we classify such languages as modelling languages. Therefore, an important question arises: when is the solver called? The answer depends on the language. The predominant approach is to call the solver every time a new constraint is added, for example in Babelsberg [FMBH15] and Turtle [GH04]. A lazy approach in Turtle++ [HK06] is to trigger the solving phase only when we read a constrained variable. An advantage of the latter method is that constraints can be added in bulk—without recalling the solver each time—while other languages need a specific statement for this feature.

A second key design decision in constraint imperative languages is to deal with unsatisfiability of a model. This is also to contrast with declarative languages where satisfiability results are delegated to external languages whereas it is solved in the same language here. The solving phase has three outcomes relevant to this class of languages:

(i) unsatisfiable in which case an exception is thrown,

(ii) exactly one solution which is the perfect scenario, and

(iii) multiple solutions where one must be chosen.

In case of unsatisfiability, it might be difficult to recover since the constraint added last might not be totally responsible for the unsatisfiability. This is why constraint imperative languages support "soft constraints", i.e. constraints that are not required to be satisfied. For example, in Turtle, they provide the annotations `weak`, `medium` and `strong` to classify constraints according to their importances. As for multiple solutions, the underlying solver tries to keep a new solution as close as possible to the previous one computed. This can be controlled using the `stay` constraints in Babelsberg. It works as follows: given a solution $\{x \mapsto [1..1]\}$, it automatically adds the soft constraint $x = 1$ with a low importance into the store.

A third dynamic aspect of constraint imperative languages is the ability to retract constraints from the store. Babelsberg provides an abstraction to retract a constraint when its declaration goes out of scope. This is realized with statements indicating if a constraint should be persistent (`always` c), only enforced the first time (`once` c) or active for a delimited block of code (`assert` c `during` p).

When we exit the scope of a constraint c, the constraint problem is updated with $\langle d, P\rangle \Delta_3(\langle d, P\rangle \Delta_3 \langle \bot_2, \{[\![c]\!]\}\rangle)$. It can help in the case of unsatisfiability if the exception flows outside the scope of an `assert/during` block.

2.4 Inference-based languages: from L_0 to L_3 $(\sqcup, \models)$

We overview backtrack-free languages dedicated to programming lattices between L_0 and L_3 with the join and entailment operators. The delta operator is virtually absent from these languages since there is no backtracking. However, these languages feature another form of nondeterminism called "don't care nondeterminism" which was described by Dijkstra with its guarded command language [Dij75]. The central concept is the nondeterministic composition of conditional statements:

```
do c1 -> p1
[] c2 -> p2
[] c3 -> p3
od
```

Among all the guards c_i that are entailed, one alternative p_i is non-deterministically selected and the others are discarded. Hence, once we committed to an alternative the program never backtracks or reconsiders its choice. However, it is important to stress that determinism of the computation is usually recovered at a higher level. Indeed, this construction is used to program systems in which every path leads to the same solution. It relates to the transition system lattice in L_3. Given a state several propagators can usually be applied and we must make a nondeterministic choice in order to progress. Nevertheless, we end up in the same final state—the top element—due to the confluence of the transition system.

2.4.1 Concurrent logic programming

Concurrent logic programming is a variant of logic programming with guards and where goals are not necessarily executed sequentially but can be suspended and interleaved. We can immediately define the main construct of this paradigm.

Definition 2.10 (Guarded Horn clause)**.** *A guarded horn clause is a Horn clause of the form*

$$H \leftarrow G_1 \wedge \ldots \wedge G_n \wedge B_1 \wedge \ldots \wedge B_m \qquad (n \geq 0, m \geq 0)$$

where G_i is an atom called the guard.

The logical interpretation is the same as with Horn clauses but its operational interpretation is different. The program is interpreted following the committed-choice semantics rather than the usual backtracking semantics of Prolog—that we consider in Section 2.5.2. This is why we classify this paradigm as being useful for programming systems up to L_3 in the lattice hierarchy. Note that,

however, this classification only pertains to combinatorial problem solving, and that concurrent logic languages might have features that languages supporting L_4 have not. Actually, in the eighties, concurrent logic programming provided a strong proposal to declaratively programming parallel and reactive systems.

Research in this field was fostered by the Japanese Fifth Generation Computer Systems (FGCS) project aiming at reworking the definition of a computer from the hardware to the software. A major goal was to program large-scale parallel machines using symbolic and knowledge-based computations (in opposition with numerical processing) [Ued17]. Therefore, the concurrent logic paradigm was deemed appropriate and chosen to cope with the parallel nature of the hardware in a declarative manner. At that time, most of the people in this community were involved in the FGCS project, and thus the peak research activity of this paradigm was from 1983 to 1992, the duration of the project. For a historical account and personal reviews of the project we refer the reader to [FKF+93] and to [Ued17] for a recent retrospective.

The language chosen as a basis for the FGCS was the Guarded Horn Clauses (GHC) [Ued85] (rebranded as Kernel Language 1 (KL1) in the project). In GHC, a guarded Horn clause is written as

H :- G_1, ..., G_n | B_1, ..., B_m

with the same meaning than in the former definition. To illustrate this paradigm, we consider a musical example making use of the guards in GHC[7].

Example 2.2. A conductor process generates a list of `1` (for play) and `0` (for silent). The players follow this list but start with a delay. The guards ensure that a player does not start if the delay is not exhausted or if the conductor requires a silent note.

```
conductor(L, C, Max) :- C =< Max | P:=(C mod 2), L = [P|R],
    C1:=C+1, conductor(R, C1, Max).
conductor(L, C, Max) :- C > Max | L = [].

player([P|R], Delay, L2) :- Delay > 0 | L2=[0|R2], D1:=Delay-1, player(R, D1, R2).
player([P|R], Delay, L2) :- P = 0     | L2=[0|R2], D1:=Delay-1, player(R, D1, R2).
player([P|R], Delay, L2) :- Delay =< 0, P = 1 | L2=[1|R2], player(R, Delay, R2).
player([], Delay, L2) :- true | L2 = [].
```

All the guards of a predicate are exclusive, and thus we do not have arbitrary choices made by the execution engine; we say that the predicate is deterministic. ⌟

Importantly, the guards are decision procedures but not propagators: they do not perform unification on their terms. For instance, the guard `P = 0` in the second predicate `player` does not unify P with 0; if P is not ground when tested, it

[7] An interpreter for GHC is still available and working today on top of `SWI-Prolog`, see `http://www.ueda.info.waseda.ac.jp/software.html`.

suspends its execution. This mechanism of suspension—central to concurrent logic programming—interprets predicates as processes by interleaving their executions. For example, we can execute the former program as follows:

```
?- ghc player(Ns, 1, Ns1), conductor(Ns, 1, 5), player(Ns, 3, Ns2).
Ns = [1, 0, 1, 0, 1],
Ns1 = [0, 0, 1, 0, 1],
Ns2 = [0, 0, 0, 0, 1].
```

where we invoke a player before the conductor. The three processes communicate on the stream *Ns* where `conductor` is the producer and the two predicates `player` are the consumers. This first player will be suspended due to *Ns* being unbound and the conductor will be executed.

At first, suspension—also called co-routining—was designed in IC-Prolog in 1979 as an attempt to relax the rigid evaluation strategy of Prolog (top-to-bottom and left-to-right evaluation of the predicates). Indeed, the former example would block with Prolog since we cannot evaluate the predicate `player` before `conductor`. Viewing co-routines as processes was first experimented in the Relational Language [CG81] in 1981. Its direct successors include Concurrent Prolog [Sha83] and PARLOG [CG83] mostly improving its synchronization mechanism, and then GHC synthesizing the research from these two to obtain a minimal and simplest language. Lastly, we must mention a restriction on the guards that led to "flat variations" of these languages: the guard must be a test predicate or the unification predicate =. This restriction prevents guards from not terminating. Henceforth, flat versions were preferred as they were simpler to implement and since deep guards were barely used in practice anyway. Actually, the FCGS's language KL1 is based on flat GHC [Ued90]. More information on the languages mentioned and examples of applications can be found in the book [Sha87] and in the reference survey [Sha89].

The dynamic aspect of backtrack-free languages

As we will see in the next section, concurrent (constraint) logic languages are monotonic functions over L_3. An aspect central to this paradigm is to be *dynamic* in the sense that the state space to explore is not fixed in advance. More precisely, in our lattice hierarchy, it means that the variable store L_2 has not a fixed cardinality during the exploration process. It is materialized in languages by constructions creating new variables during the execution. In concurrent logic languages, it is hidden in the bindings created when instantiating new predicates.

It explains why these languages stop at L_3: the inference process might not terminate. Hence, even if it provides support for L_4, the question is when should it stop the inference to add new nodes in L_4? We consider some of the proposals in Section 2.5. Moreover, since it possibly describes infinite computation, the search tree might grow indefinitely and not be bounded in space. Note that this problem of unbounded memory is usually avoided in L_3 by removing the variables

$$
\begin{array}{llr}
\langle \textit{Program} \rangle & ::= \ (t(x_1, \ldots, x_n) \text{ :- } p)+ & \text{(process definition)} \\
\langle p,\ q,\ \ldots \rangle & ::= \ p \ \parallel \ q & \text{(parallel)} \\
& \mid \ \sum_{i \in n}^{n} \textbf{ask } (c_i) \ \textbf{ then } \ p_i & \text{(guarded sum)} \\
& \mid \ \textbf{tell}(\text{c}) & \text{(tell)} \\
& \mid \ \exists x.p & \text{(hiding operator)} \\
& \mid \ t(x_1, \ldots, x_n) & \text{(call)}
\end{array}
$$

Figure 2.2: Syntax of CCP

that do not participate in the computation anymore—for instance using a garbage collector[8]. In comparison, removing a node from L_4 has a more crucial impact since it also removes the chance to find a solution. To sum up, computations in L_4 are usually a memory and time bounded activity whereas concurrent processes in L_3 can describe infinite computation.

2.4.2 Concurrent constraint programming

The guards and committed choice in concurrent logic programs were not well understood from a logic point of view and lacked of a well-defined semantics. This problem was solved in [Mah87] by defining a semantics based on constraints and using the entailment operator $\models$ for checking the guards. An additional insight was to consider the entailment as a synchronization operator between the predicates. Based on these results, Saraswat and Rinard developed concurrent constraint programming (CCP) [SR89], a constraint-based process calculus. To quote [Ued17]: "Concurrent Constraint Programming tends to be considered as marriage of Concurrent Logic Programming and Constraint Logic Programming, but it emerged as the formalization of the former inspired by the latter."

Overview and syntax

A program in CCP is a set of processes communicating through a shared global constraint store. They interact in this store with two primitives: $tell(c)$ for adding a constraint c into the store and $ask(c)$ for asking if c can be deduced from the store. Concurrency is treated by requiring the store to grow *monotonically*, i.e. removal of information is not permitted. For example, if it initially contains `x > 5`, adding `x > 2` will not change the store and `x < 2` will fail the whole computation due to contradictory information.

The syntax of the calculus is shown in Figure 2.2; the notation is inspired by [BPP95] where p, q are processes, x_i a variable name, c a constraint and t a

[8] We can see the garbage collector as a function using the meet operator over L_2 removing variables that are only referenced by entailed propagators in L_3.

process name. A well-formed program is a set of process definitions with distinct names. The different operators have the following meaning:

- p `||` q executes the processes p and q in parallel.
- $\sum_{i \in n}^{n}$ `ask(`c_i`) then` p_i is the guarded committed-choice operator: one process p_i among the ones with an entailed guard c_i is executed.
- $\exists x.p$ declares a new variable x accessible from the process p only.
- $tell(c)$ is used to add a constraint c into the global store.
- $t(x_1, \ldots, x_n)$ invokes the process declared with the name t.

In this concurrent framework, the predicates become processes and their order has no more importance. This is why CCP adopts a syntax closer to usual process calculi such as Hoare's communicating sequential processes (CSP) formalism [Hoa78].

An important trait of CCP is to leave abstract the concrete syntax and semantics of the underlying constraint system. This strategy conveniently avoids the operational complexity of constraint solving, it helps to stay generic and to focus on the language design. As we will see shortly, CCP paved the way to many research for instantiating this paradigm to various constraint systems [Sar93]. More recently, constraint systems have been thoroughly studied in the context of SMT theories (introduced in Section 2.2.4). Similarly to SMT, we formally consider a constraint system as a first-order logic theory.

On the relation to concurrent logic programming

A first constraint system is the Herbrand constraint system, or the theory of equality and uninterpreted functions (Definition 2.8). It justifies that concurrent logic languages can be seen as instantiated versions of CCP. For instance, using this theory and two interpreted arithmetic functions, we can transform the former GHC program into an equivalent CCP program.

Example 2.3. Consider the transformation of the first predicate `conductor` of Example 2.2:

```
conductor(L, C, Max) :- ask(C =< Max) then ∃P,C1,R.
                           (  tell(P=(C mod 2))
                           || tell(L = [P|R])
                           || tell(C1 = C+1)
                           || conductor(R, C1, Max))
                        +
                        ask(C > Max) then tell(L = []).
```

⌟

Using the committed-choice sum operator, we can simulate the guarded Horn clauses of GHC. In the example, the process contains the two different variants of the logic predicate `conductor` composed with the sum operator. The atoms of the clause are composed in CCP using the parallel operator instead of the comma operator p, q in logic programs. Finally, the implicit creation of variables in logic programs is made explicit through the existential operator and new bindings are added in the store with equality constraints. The unification algorithm is the internal solving algorithm of the Herbrand constraint system. From these observations, it appears more clearly that concurrent logic programming is CCP instantiated to the Herbrand constraint system.

In the following, we give a novel reinterpretation of the semantics of CCP using the lattice framework defined in Chapter 1. We consider that it is adaptable (by specialization) to obtain the semantics of (flat) concurrent logic languages. More information on the relation between both paradigms can be found in the survey [dBP94].

Lattice-based semantics of CCP

We define the semantics of CCP in the settings of the lattice framework. A reduction rule has the form

$$D \vdash (\langle d, P\rangle, p) \rightarrow (\langle d', P'\rangle, p')$$

where D is the set of processes' definitions, the store $\langle d, P\rangle$ is defined over the lattice of CSPs L_3 and p is the current CCP process.

The semantics rules are given in Figure 2.3. We use the process `nothing` as an alias for the empty sum representing the empty process.

- The rules *par-left* and *par-right* respectively reduce the left and right process in the parallel statement; their executions can be interleaved.
- In *atomic-tell*, we assign two responsibilities to `tell(`c`)`: (i) joining the constraint c into the store, and (ii) reducing the store to the top element of its transition system, i.e. performing the propagation on the store.
- Our treatment of the existential operator in the rule *hiding* is quite different to what is usually found in the literature. Usually, it delegates the treatment of the existential operator to the theory itself. We take a more operational approach by considering $\exists x$ as the creation of a new variable in L_2 at a fresh location. The function *alloc* (Definition 1.26) allocates the new variable at the next available location, retrieved with *maxloc*. In addition, we use the substitution operator to replace every syntactic occurrence of the variable by its location in the store. The existential operator is central to building dynamic state space as we discussed in the previous section.

$$\text{par-left}\frac{D \vdash (\langle d, P\rangle, p) \rightarrow (\langle d', P'\rangle, p')}{D \vdash (\langle d, P\rangle, p \texttt{ || } q) \rightarrow (\langle d', P'\rangle, p' \texttt{ || } q)}$$

$$\text{par-right}\frac{D \vdash (\langle d, P\rangle, q) \rightarrow (\langle d', P'\rangle, q')}{D \vdash (\langle d, P\rangle, p \texttt{ || } q) \rightarrow (\langle d', P'\rangle, p \texttt{ || } q')}$$

$$\text{atomic-tell}\frac{\langle d, P \cup [\![c]\!]\rangle \Rightarrow \langle d', P'\rangle}{D \vdash (\langle d, P\rangle, \texttt{tell}(c)) \rightarrow (\langle d', P'\rangle, \texttt{nothing})}$$

$$\text{hiding}\frac{d' = alloc(d) \qquad D \vdash (\langle d', P\rangle, p[x \rightarrow maxloc(d')]) \rightarrow (\langle d'', P''\rangle, p')}{D \vdash (\langle d, P\rangle, \exists x.p) \rightarrow (\langle d'', P''\rangle, p')}$$

$$\text{call}\frac{D, t(y_1, \ldots, y_n)\texttt{:-}p \vdash (\langle d, P\rangle, p[y_1 \rightarrow x_1, \ldots, y_n \rightarrow x_n]) \rightarrow (\langle d', P'\rangle, p')}{D, t(y_1, \ldots, y_n)\texttt{:-}p \vdash (\langle d, P\rangle, t(x_1, \ldots, x_n)) \rightarrow (\langle d', P'\rangle, p')}$$

$$\text{choice}\frac{Sol_{3to2}(\langle d, [\![c_i]\!]\rangle) \models_{\mathscr{A}(Sol_2)} Sol_{3to2}(\langle d, P\rangle) \qquad D \vdash (\langle d, P\rangle, p_i) \rightarrow (\langle d', P'\rangle, p'_i)}{D \vdash (\langle d, P\rangle, \sum_{i \in n}^{n} \texttt{ask}(c_i) \texttt{ do } p_i) \rightarrow (\langle d', P'\rangle, p'_i)}$$

Figure 2.3: Lattice-based semantics rules of CCP.

- Next, the rule *call* substitutes a process call with its definition. We retrieve the definition from the context D and substitutes the parameters occurring in the body by the arguments of the call.
- Finally, the rule *choice* formalizes the commitment to one alternative for which the guard is entailed in the current store. Intuitively, a guard is entailed if it does not remove solutions from the current store; in other words, if it is a redundant constraint. Formally, it is given by the order over the antichain lattice $\mathscr{A}(Sol_2)$ over the solutions of L_2. We explain this operator in more depth just thereafter because it is central to understand the discrepancy between the CCP's theory and its usage in programming languages.

Summing up, the store is a L_3 structure where `tell` is the join operation in L_3, `ask` is the entailment over $\mathscr{A}(Sol_2)$ and the hiding operator $\exists$ is the join operation in L_2.

The entailment paradox

In the committed choice rule, the entailment is represented by the expression

$$Sol_{3to2}(\langle d, [\![c_i]\!]\rangle) \models_{\mathscr{A}(Sol_2)} Sol_{3to2}(\langle d, P\rangle)$$

specifying that the set of solutions in $\langle d, P\rangle$ is contained in the set of solutions given by the guard c_i. It could be equivalently written

$$Sol_{3to2}(\langle d, P\rangle) = Sol_{3to2}(\langle d, P \cup [\![c_i]\!]\rangle) \tag{2.1}$$

where we ensure that the guard does not constrain the set of solutions of the initial problem. To illustrate better this operator, we give an example.

Example 2.4 (Entailment of a guard)**.** Consider the following CSP:

$$A = \langle\{x \mapsto [0..2], y \mapsto [0..2]\}, \{[\![x = 0]\!], [\![x < y]\!]\}\rangle$$

Then we can ask: is the constraint $y > 1$ entailed in the store A? According to (2.1), we must compute the sets of solutions of A and of the constraint $y > 1$:

$$\begin{array}{l} Sol_{3to2}(A) \mapsto \{\{x \mapsto 0, y \mapsto 1\}, \{x \mapsto 0, y \mapsto 2\}\} \\ Sol_{3to2}(\langle\{x \mapsto [0..2], y \mapsto [0..2]\}, \{[\![y > 1]\!]\}\rangle) \mapsto \\ \quad \{\{x \mapsto 0, y \mapsto 2\}, \{x \mapsto 1, y \mapsto 2\}, \{x \mapsto 2, y \mapsto 2\}\} \end{array}$$

and thus we cannot yet deduce $y > 1$ from the current problem since the solution set of $y > 1$ does not contain $\{x \mapsto 0, y \mapsto 1\}$ which is a solution of A. However, this constraint can be entailed later if we add information into the store that restricts its set of solutions. ⌟

A paradox emerges as soon as we obtain a non-convex CSP (Definition 1.35) during a computation. As we saw in Chapter 1, solving a non-convex CSP requires a function over the lattice L_4. However, CCP's programs are functions over L_3. The paradox is that implementing the entailment turns out to be itself a constraint problem. Moreover, due to non-convex CSPs, this constraint problem needs to be solved in L_4 or higher whereas CCP is actually designed to manipulate structures in L_3. The problem being that solving a problem in L_4 is usually much more costly than solving one in L_3. Based on this paradox, we can enunciate a simple principle when designing a language with operators mapping to the lattice hierarchy.

Principle 2.1 (Lattice hierarchy for language design)**.** *Given a language defining operators over a lattice L_i, these operators must be internally implementable by a function over the lattices L_j with $j \leq i$.*

This paradox, whilst happening in the full generality of the CCP calculus, can be avoided in two different ways when instantiating CCP to a constraint system:

(i) Using convex constraint systems.

(ii) Relaxing the entailment rule.

Firstly, among convex constraint systems, we already mentioned Herbrand terms that are directly useful for programming in the concurrent logic paradigm. The Herbrand constraint system was notably extended to records in languages merging the logic and functional paradigms. LIFE is a pioneer functional logic language which builds on the order-sorted feature (OSF) constraint system [AKP93] for manipulating records in logic languages. This idea was later used in Oz, a language based on CCP, which proposed efficient constraint checking and entailment for records structures [RMS96]. Despite their interests, functional logic languages fall out of scope in this survey. However it is interesting to consult [AH10] for an introduction to the field, the survey [Han07] for a more detailed presentation and [Han14, Hof11] for functional logic languages in the larger context of multi-paradigms languages.

Secondly, when taking the case of a non-convex constraint system, the entailment check must be relaxed to a computationally cheaper operation. In the next section we present a non-convex constraint system for finite domains named FD [VHSD91]. When implementing CCP with this constraint system [VHSD98], they use the domain entailment (Section 1.4.2) which checks a constraint against the domain in L_2 only (and not the full CSP in L_3). The choice rule is modified as follows:

$$\text{choice-dom}\ \frac{d \models_d [\![c_i]\!] \qquad D \vdash (\langle d, P\rangle, p_i) \rightarrow (\langle d', P'\rangle, p'_i)}{D \vdash (\langle d, P\rangle, \sum_{i\in n}^{n} \mathsf{ask}(c_i)\ \texttt{do}\ p_i) \rightarrow (\langle d', P'\rangle, p'_i)}$$

The domain entailment can be computed using the lattice L_3 only. It provides a safe relaxation of the “full entailment”: anything entailed with the domain entailment is also entailed with the full entailment. However, the converse is not true, computations that are suspended with this entailment might be executable in the full variant. Finally, we must mention that there exists a few solving algorithms for the entailment in L_3 [DdlBS04, DJ12]. Of course, these algorithms are defined over the lattice L_4 and the entailment request is computed while solving the CSP.

Finite domains and indexicals constraint system

One of the most influential constraint systems is the theory of finite domains FD and indexicals as first introduced in [VHSD91]. An *indexical* is an unary constraint over a variable enforcing its domain to be in a specific interval. The syntax of this constraint system is given in Figure 2.4. It is interesting because, compared to other theories, it makes the underlying event system of the propagation algorithm explicit.

Example 2.5 (Indexical $x < y$)**.** To illustrate this claim, consider the indexical version of the propagation function of the constraint $x < y$.

$$\begin{aligned} [\![x < y]\!] \ &\stackrel{\text{def}}{=} y\ \texttt{in}\ (\mathsf{Min}(x) + 1)..\texttt{inf} \\ &\land\ x\ \texttt{in}\ 0..(\mathsf{Max}(y) - 1) \end{aligned}$$

$$\langle t \rangle ::= X \mid n \mid t + t \mid t - t \mid t * t \mid t \; mod \; t \mid t \; / \; t$$
$$\mid \quad min(X) \mid max(X) \mid val(X)$$

$$\langle r \rangle ::= t_1..t_2 \quad \text{(range)}$$
$$\mid \quad r_1 \; : \; r_2 \quad \text{(union)}$$
$$\mid \quad r_1 \; \& \; r_2 \quad \text{(intersection)}$$
$$\mid \quad -r \quad \text{(complementarity)}$$
$$\mid \quad r + t \quad \text{(pointwise addition)}$$
$$\mid \quad r \; mod \; t \quad \text{(pointwise modulo)}$$
$$\mid \quad dom(X)$$

$$\langle c \rangle ::= X \; in \; r$$

Figure 2.4: Syntax of the FD constraint system with indexicals.

where $Min(x)$ and $Max(x)$ are functions mapping to the lower and upper bounds of the variable x with $t..t'$ describing an interval with bounds between t and t'. In the implementation, the propagation engine can analyse Min and Max in the definition of a constraint to know under what event the propagator might impact the domains. Here, it is useful to reschedule $x < y$ only if the lower bound of x or the upper bound of y have changed. This is a language-version of the Example 1.16 which formalized this propagator within the lattice framework. Hence, indexicals are operators over the lattice L_2 that can be used to program propagators in L_3. ⌟

We mentioned that propagators must fulfil two properties: being contracting and sound (Definition 1.4). Indexicals can be automatically checked to fulfil these two properties, and even more strongly, we can prove an indexical to be monotonic or anti-monotonic [CCD94]. Monotonicity means that the interval of a variable can only be reduced (monotone function over L_2) and anti-monotonicity that it can only increase (monotone function over L_2 with its the order reversed). In the context of CCP and more precisely the tell and ask operators, an important insight given by the indexicals is that tell constraints must be monotonic functions and ask constraints must be anti-monotonic functions in order to preserve the overall monotonicity of the computation.

Example 2.6 (Indexicals in CCP)**.** Consider a CCP process that ensures that whenever a variable Y becomes greater or equal to X then Y becomes automatically strictly greater than X.

```
if_geq_then_gt(X,Y) :- ask(Y in Max(X)..inf) then tell(Y in (Min(Y) + 1)..Max(Y))
```

The ask constraint is anti-monotonic: once Y is in the range `Max(X)..inf`, it stays entailed for the rest of the computation since the range will evolve anti-

monotonically (the range will increase), and that Y will evolve monotonically (its range will decrease). ⌟

From this example, we notice that an indexical constraint is only defined over L_2 since it is defined with operators solely defined on L_2. In particular, the events generated by the operators Min and Max are exactly `lbc` and `ubc` defined in Example 1.15.

Indexicals are applicable beyond the scope of CCP only and are actively used in constraint logic programming which is treated in Section 2.5.2.

Extensions of CCP

To conclude this overview of CCP, we discuss several shortcoming and associated extensions to the semantics of CCP.

Inconsistent store A particular case, not explicitly treated in the semantics rules, is the unsatisfiability of the store. Indeed, the tell operator can generate an inconsistent store, or more formally $\langle d, P\rangle \in Fail_3$. In this case, we have $Sol_{3to2}(\langle d, P\rangle) = \{\}$ and thus it is entailed by any element since $\{\}$ is the bottom element of the antichain lattice $\mathscr{A}$. In practice, this case must be treated with care; for example a rule handling this situation in tell can be added:

$$\text{tell-fail}\frac{\langle d, P \cup [\![c]\!]\rangle \Rightarrow \langle d', P'\rangle \qquad ssd(\langle d', P'\rangle) = false}{D \vdash (\langle d, P\rangle, \mathtt{tell}(c)) \rightarrow (\langle d, P\rangle, \mathtt{fail})}$$

where `fail` is a process representing a failed state. Also note that the constraint is not added into the store.

Eventual tell A problem with the atomic tell occurs in distributed applications: it requires to coordinate the processes to temporarily suspend their activities in order to atomically solve the store. Since tell is a very basic and frequent operation, this would entail the processes to synchronize all the time which is not manageable in a distributed setting. Hence, it motivated an important variant of this rule, called eventual tell, which delays the inference step. This rule is written as

$$\text{eventual-tell}\frac{}{D \vdash (\langle d, P\rangle, \mathtt{tell}(c)) \rightarrow (\langle d, P \cup [\![c]\!]\rangle, \mathtt{nothing})}$$

The constraint is just added into the store without being solved right away. For example, propagation can be triggered on an entailment request or through more evolved mechanism (see Chapter 7 in [Sar93]). However, it complexifies the case where the store becomes inconsistent and to recover from such a situation.

Constraint-based concurrency Since its inception, CCP has been extended with various constructions borrowed from other paradigms and languages. It forms a theoretical line of research closer to concurrency theory than to practical programming languages. This is why some authors prefer the term "constraint-based concurrency" instead of "concurrent constraint programming" when speaking about CCP in this context [Ued17]. We only review one relevant extension of CCP for the rest of this work; a survey on constraint-based concurrent languages is available in [ORV13].

Anticipating a bit on Chapter 3, an important extension is temporal CCP (abbreviated as TCC) from Saraswat [SJG94, SGJ14]. It imports the notion of logical time from synchronous languages into CCP. The motivation is to periodically forget about some information that are not necessary anymore to the computation. In effect, it proposes an implicit meet operator over the constraint store which is invoked between two time points—called instants. Moreover, by dividing the computation into bounded instants, we can ask about negative information: is a constraint not entailed at some point in time? This is not possible in CCP since a constraint not entailed now might become entailed later.

2.5 Bridging L_3 and L_4 $(\sqcup, \models, \Delta)$

In the previous section, we saw that languages for L_3 are suited to program infinite computation. Seemingly, they appear to be incompatible with the layer L_4 of the hierarchy since it often implies a certain notion of finiteness and exhaustiveness of the state space. We review two important attempts to bridge these two layers.

On the one hand, we have languages that try to re-integrate backtracking into the concurrent (constraint) logic paradigm; they are mostly based on the Andorra principle explained below.

On the other hand, constraint logic programming (CLP) generalizes logic programming by allowing users to program over L_3 with any kind of constraints—and not just the Herbrand constraint store. In comparison, CLP provides backtracking without concurrency whereas CCP provides concurrency without backtracking.

In short, Andorra-based languages combine L_4 operators with L_3 languages while CLP languages try to do the opposite (by generalizing L_3).

2.5.1 Andorra-based languages

Andorra principle

When it comes to blend don't-care and don't-know nondeterminism into a single logic language, virtually every proposition has its root into the *Andorra principle* enunciated by David Warren in 1987[9] and pioneered by P-Prolog [YA86]. Basically, it says that

[9] According to [Har90], it was during a GigaLips meeting in Stockholm in June 1987.

(i) every goal that is determinate—only one predicate is applicable to reducing this goal—should be selected first, and

(ii) when only nondeterminate goals remain, one is selected by creating a choice point and is backtracked to this point in case of failure.

The first rule refers to don't-care nondeterminism and the second to don't-know nondeterminism. In fact, this apparent simple principle occurs under many different forms. One of them has already been introduced: we reach the top element of L_3 by inference until every propagator is at a fixed point (rule (i)), and then we split the search space in L_4 (rule (ii)). In this case, the notion of "determinacy" is clear since it is precisely defined by the propagation algorithm. However, when reducing a logic goal that has multiple predicate definitions with the same name, we must partially evaluate the different choices in order to verify if one or several is applicable. The question is how far should we evaluate the goal? To illustrate different forms of determinacy, we consider the following logic program.

```
first([], F, N, []).
first(L, F, N, L2) :- N == F, L2 = [].
first([H|L], F, N, L2) :- N < F, N2 is N + 1, first(L, F, N2, L3), L2 = [H|L3].
```

From these definitions, we know that `first([],0,0,L)` can only be unified by the first predicate. It is a weak form of determinacy that tries to unify the goal with every predicate head. If only one clause is applicable, then the goal is considered determinate and we reduce it first. Another example is `first([a],1,0,L)` where, in this case, the condition `N == F` and `N < F` must be evaluated to realize that only the third clause is applicable. This leads to a stronger form of determinacy check. Accordingly, evaluating the different clauses more or less deeply leads to different notions of determinacy. In practice, three early Andorra-based languages—namely Andorra-I [CWY91], Andorra Prolog [Har90] and Pandora [Bah93]—all implemented determinacy by trying to perform unification and evaluating the built-in predicates appearing before any user-defined predicates.

The motivation behind the determinacy check was also to perform automatic parallelization of logic programs. Indeed, an orthogonal interpretation of the Andorra principle is to see a logic program as an "and-or tree" where the rule (i) generates "and-nodes" and rule (ii) generates "or-nodes". For example, "and-parallelism" executes in parallel determinacy checks and independent goals which do not bind identical variables. In contrast, "or-parallelism" evaluates in parallel nondeterminate clauses when instantiating a goal. We do not consider the various techniques of parallel logic languages; a survey can be found in [GPA+01] and an overview in [SC00]. Today's constraint solvers are mostly performing or-parallelism since it is easier to implement and offer good efficiency [Sch00].

Deep guards and encapsulation

Committed-choice deep guards coupled with determinacy checks led to an important feature called *encapsulated search*. Historically, it starts with Andorra

Kernel Language (AKL)—succeeding to Andorra Prolog[10]—proposing a practical language integrating deep guards, constraints and concurrency [JH91, JUIS94]. We explain how their combinations provide encapsulated search.

Deep guards were first considered in concurrent logic programming and then discontinued in favour of flat guards, easier to implement while retaining enough expressiveness (see Section 2.4.1). At that time, one of the challenges is to evaluate deep guards without modifying the current binding store. Indeed, if the guard is disentailed, the store must be left unchanged. Therefore, we ensure that the evaluation of the guard is independent from the global store (or is *quiet* in the terminology at this time). Independence is a property ensuring that no variable external to the guard can be bound to a new value. Moreover, if the guards are independent, they can be evaluated concurrently.

The idea of encapsulated search in AKL is to solve a CSP inside a guard. The semantics of guards ensures that even nondeterministic computation stays encapsulated in the guard without disrupting the whole computation. In case of an unsatisfiable CSP, the guard is disentailed and thus the body of the clause is not executed. In case of satisfiability, the result of the encapsulated computation can be retrieved in a variable local to the clause. Moreover, using a primitive called `bagof/3`, users can retrieved all the solutions of the CSP into a list of variables.

Logic programs within the lattice hierarchy

Anticipating a little on Section 2.5.3, where we formalize the semantics of constraint logic languages, it is worth understanding now the theoretical differences between propagators and goals from the perspective of the lattice hierarchy. We recall that a propagator is a function from L_2 to L_2 and a CCP process is a function from L_3 to L_3. Logic programs are yet another entity. On the one hand, a logic program generates new predicates whereas a propagator does not generate new propagators. On the other hand, whilst CCP processes also generate new processes (by unfolding process definitions), they cannot be backtracked and thus are not part of the data structure. In Section 2.5.3, we will formally describe the structure of logic programs as a constraint store in L_3 endowed with the predicates to instantiate.

Consequently, the notion of deep guards finds a hierarchical explanation: it lifts the computation one level higher in the hierarchy. Encapsulated search in a deep guard is a function over L_3 which is interpreted in a L_4 environment since it searches for the solution to a CSP—the global store being copied when entering the guard in order to satisfy the independence condition. Therefore the program calling this guard is defined over L_5 since, intuitively, by calling several guards, it can explore several trees. To sum up, a flat guard is a function over L_2 and a deep guard is a function over (at least) L_3. Generalizing the notion of constraint

[10]More precisely, an intermediate formulation between Andorra Prolog and AKL, called Kernel Andorra Prolog [HJ90], first generalized the Andorra model to constraints and AKL is a practical instantiation of this framework.

entailment to logic programs, they suggested in AKL to refer about "stability" of a guard, which means that the guard is either entailed (flat guard) or cannot be reduced anymore (deep guard).

Moreover, the size of the hierarchy is delimited by the number of nested deep guards generated by the program. Nested deep guards were explored in the context of Oz [Smo94], succeeding to AKL, where an L_3 structure and a set of threads (i.e. functions computing over L_3) form a *computation space* [Sch02]. A computation space encapsulates the search, can be nested into another space, and provides methods to query its stability.

2.5.2 Constraint logic programming

Constraint logic programming (CLP) generalizes logic programming to arbitrary constraint systems in a very similar way to CCP—note that both paradigms emerged around the same time. The road to CLP was first undertaken with Prolog II, in the beginning of the eighties, by recognizing the unification algorithm as an instance of constraint solving [Col85]. In this same version, they also proposed the predicate `dif/2` modelling the constraint $\neq$. Additional arithmetic constraints were added later into Prolog III [Col90]. Instead of adding constraint predicates in an *ad-hoc* manner, Jaffar and Lassez formally extended the uninterpreted theory underlying logic programming with symbols that could be interpreted over a fixed domain [JL87]. In comparison with pure Horn clauses, the problem is captured more clearly and constraint solving becomes orthogonal to the modelling activity. Among the early CLP languages, there are CLP(R) described in [JMSY92] for solving constraints over real numbers, CHIP specialized in finite domains [VH89] and CLP(BNR) combining solvers on boolean, natural and real numbers [Old93]. Modern CLP languages include solvers on several domains such as real, rational, boolean and finite domains. To be clear, finite domains generally referred to integers domains or any domain that is isomorphic to a finite subset of $\mathbb{Z}$—the boolean domain is a specialization of finite domains. Real and rational domains are easier to integrate into Prolog since CSP generated by arithmetic constraints over these domains are usually convex. On the other hand, arithmetic over boolean and integer domains form non-convex problems and thus need a search step to be solvable. This is the most challenging part since the constraint solving part and Prolog backtracking are closely linked. This is why, a large part of the research on CLP is actually specialized to the finite domains, and this is the main topic of this section.

We proceed step-by-step, and almost chronologically, to the features added into plain logic programming to obtain the CLP paradigm. In particular, a very clear resource on this matter is the book [AW07]. After considering the search part, we take a look on more recent developments for programming propagators efficiently inside CLP instead of relying on external solvers. We devote the Section 2.5.3 to the formal semantics of CLP using lattice theory.

Constraint solving in Prolog

Constraint solving in Prolog III and `CLP(R)` was tackled using algorithms over symbolic terms. There is no search step involved with the solving and, in fact, the initial CLP theory of [JL87] requires the constraint system to be *satisfaction complete*—or convex in our terminology. For non-convex system, a generate-and-test algorithm can be implemented using the backtracking strategy of Prolog.

Example 2.7 (Decreasing sequence)**.** A well-known global constraint is to constrain a sequence of variables to be decreasing, mathematically we write $x_1 \geq x_2 \geq \ldots \geq x_n$. In Prolog, we can use arithmetic predicates to ensure this behavior on a list:

```
geq(X, Y) :- X >= Y.

decreasing([]) :- true.
decreasing([Y]) :- true.
decreasing([X, Y | R]) :- geq(X, Y), decreasing([Y | R]).
```

However, if we try to call this predicate directly with `decreasing([X,Y,Z])`, we obtain an error indicating that the variables are not sufficiently instantiated. Indeed, when the execution engine encounters the first constraint `X >= Y`, X and Y are non-instantiated, and there is no built-in mechanism for enumerating their values. The solution is to perform the search by ourselves:

```
min_dom([X | D], X).
min_dom([X | D], Val) :- min_dom(D, Val).

enumerate([X | R], Dom) :- min_dom(Dom, X), enumerate(R, Dom).
enumerate([], Dom) :- true.

search(Vars) :- enumerate(Vars, [1,2,3]), decreasing(Vars).
```

We use the most basic branching strategy where the values of the variables are enumerated in the order of appearance—if the list is ordered, it starts at the lowest value in the domain of the variable. For simplicity of the presentation, we use the same domain $\{1, 2, 3\}$ for each variable. Note that it is the predicate `min_dom` that introduces the choice point: either we select the head of the list or try the next value, and that recursively until the list is empty. Importantly, `decreasing` is placed after `enumerate`, otherwise we would again have an instantiation error. ⌟

A problem occurring in this example is the inefficiency of the constraint checking. A key point being that we do not need to instantiate all the variables to notice that the partial assignment $X = 1, Y = 2$ will not lead to an decreasing sequence. This problem, arising with the classic semantics of Prolog, is one of the reasons that led to co-routining[11]. The central idea of co-routining is to suspend a goal until an event happens. In the case of Prolog II, the predicate `freeze/2` suspends a goal until a variable is instantiated. We can reformulate the predicate `geq` using suspension:

```
geq(X, Y) :- freeze(X, freeze(Y, X >= Y)).
```

The constraint `X >= Y` is now executed only when the variables X and Y are ground. Furthermore, the predicate does not block the whole execution and is put on a list aside until its watched variables—through `freeze`—become ground. It means that we can reverse the predicates `enumerate` and `decreasing` in our former example. This leads to an improved algorithm where, as soon as a constraint is not satisfied, the system backtracks; we do not need to wait for full assignment.

Constraint solving in CLP

When developing the propagate-and-search algorithm in Section 1.1.2, we gave two roles to propagators: deciding if the constraint is entailed and filtering unsatisfiable values out of the domains. In the Prolog jargon, propagators fulfilling the first role only are referred to as *passive constraints* and those fulfilling both roles are referred to as *active constraints*. This first role is efficiently supported in logic programming by the suspension mechanism. The second role is much more of a problem in classic Prolog.

As highlighted in [JM94], it is possible to encode domains and a propagation engine inside Prolog but it complexifies the program and represents a significant work load on the programmer's shoulders. For instance, we must encode the domains as list of values using Herbrand terms. However, it is not possible to remove the values from this list since it would be anti-monotonic according to the Herbrand constraint system. The solution is to create a new list each time we need to remove a value from this list. Besides being inefficient, it shows the inadequacy of Herbrand terms to deal with finite domains. With Herbrand terms, monotonicity is preserved by expanding terms—more and more bindings are generated—whereas in finite domains it is preserved by narrowing domains—less and less values are present. The solution, proposed in [HD86], was to combine terms and domains in a same logic program. Once domains were added to Prolog, it was natural to add propagation over these domains, and as a matter of fact, one year later was born CHIP (*Constraint Handling in Prolog*) [VH89, VH91], the first CLP language to include propagation. However, CHIP defines the propagation directly inside the execution engine, leaving the constraint system as a "black box". It prevents users from programming their own propagators which can be instrumental in solving a problem efficiently.

The solution, initially designed in the context of CCP (see Section 2.4.2), was to use the finite domains and indexicals constraint system in the context of CLP. This led to the development of the language CLP(FD) [DC93, CD96] which integrated very efficiently the indexicals into the existing Prolog compilation scheme.

Example 2.8 (Decreasing with indexicals)**.** The predicate `geq` of the former example is simply modified as:

[11] Curiously, we pointed out in the former section that this is the exact same mechanism which is at the premises of the concurrent logic paradigm.

```
geq(X, Y) :- X #>= Y.
```

and `decreasing` is left unchanged. The constraint `#>=` is a short-cut for the corresponding indexical, similarly defined to the one in Example 2.5. Indexicals are not standardized and, as we will see below, implementations support them in different ways. ⌟

As we already know, propagation alone is not sufficient for solving every problem and we must perform a search step.

Example 2.9 (Splitting on the middle value)**.** The following code in GNU Prolog [DAC12] shows a branching strategy selecting variables from left to right and splitting domains in the middle.

```
enumerate([X | R]) :- split_mid(X), enumerate(R).
enumerate([]) :- true.

middle(X, M) :-
  fd_min(X, Min),
  fd_max(X, Max),
  M is (Min + Max) // 2.

split_mid(X) :- integer(X), !.
split_mid(X) :-
  middle(X, M),
  (
    X #=< M
  ;
    X #> M
  ),
  split_mid(X).
```

Enumerating the variables of the problem is done identically to the former example. The predicate `split_mid/1` succeeds only once the variable is bound to a single integer which is verified with `integer/1`. Otherwise, we create two branches by constraining the variable to be less or equal (resp. greater) than its middle value. In the predicate `middle`, the built-in `fd_min/2` introspects the domain of the variable X and bind Min to its lower bound—similarly with `fd_max/2` for the upper bound. This is next used to compute the middle value of X. ⌟

This example gives two important insights on the implementation of CLP(FD)'s languages:

- Theoretically, `fd_min(X, Min)` (similarly for `fd_max/2`) is not a monotonic predicate, this predicate is only satisfied when the lower bound of X is equal to Min, which is the case only for a fraction of the execution.
- In practice, domains are usually implemented with a mechanism called *attributed variables* [Hol92] which attaches arbitrary information to a variable.

In the example, we rely on built-in predicates (e.g. `fd_min/2`) to access attributes' fields. The problem with attributed variables is to delegate the semantics of unification—on these particular variables—to user-defined predicates. This is a strength as it provides additional flexibility to Prolog, for example in this case going beyond the unification of Herbrand terms. However, it seems somewhat *ad-hoc* to Prolog semantics and weakens the initial theoretical proposal of CLP since the implementation does not reflect the theory.

Beyond these two remarks, even if indexicals were perfectly integrated, there are still limited in at least two ways:

- Despite that the propagation itself is expressed in "white-box", the event system is built-in inside the language and limited to a small set of events. For example, $Max(X)$ schedules the propagator when the upper bound of X is changed. However, for a matter of efficiency, it is sometimes necessary to react to more complex events.
- It does not support arrays of variables which is central to programming global constraints.

These problems are not well-solved yet: there exists almost as many proposals as there are CLP systems. Global constraints and reacting to events can be programmed in a low-level language like C (e.g. SICStus [CM12]), in a dedicated language (e.g. GNU Prolog [DC01]), and with interfaces for connecting external solvers (e.g. Eclipse [SS12]). Other systems only propose non extensible built-in propagators (e.g. SWI-Prolog [WSTL12]). In the next section, we outline major event systems' extensions to CLP. Two languages, tightly relevant to CLP, that can be used to program global constraints are action-rules extending B-Prolog [Zho06] and constraint handling rules (CHR) [Frü98]. Actually, CHR is a research field in its own for programming constraint systems (in L_3) that gets its roots into various paradigms including CLP. Hence, we leave the analysis of CHR in the context of the lattice hierarchy for future works.

Programming event systems

GNU Prolog, as the successor of CLP(FD), extends indexicals with arbitrary event conditions and an operator to retract indexicals [DAC12]. In the lattice hierarchy, event conditions correspond to the entailment over L_0 and removal of indexicals to the delta operator over the constraint store in L_3. As shown in [DAC12], the combination of these two operators can be used to replace a set of propagators by stronger ones when a condition becomes entailed. Hence, the meet operator is used in a "safe way" in the sense that it does not remove solutions from the initial CSP. However, this kind of verification is left to implementers since no automatic checking is done.

In Eclipse, successor of several pioneering CLP systems including CHIP, events are managed by extending the suspension mechanism (e.g. `freeze/2`) [AW07, SS12]. The built-in predicate `suspend(Pred, Prio, Events)` wakes up the predicate `Pred` when one of the events in the list `Events` is triggered. In case several predicates wake up on the same event, the ones with the lowest priority *Prio* are first executed. The event system of Eclipse is built around a general event model comprising three events in the totally ordered set {`inst`, `bound`, `constrained`}. The two first events are proper to the Herbrand constraint system while `constrained` extends the event system to arbitrary CLP domain—`constrained` is generated on any domain change. We distinguish between `inst` for a variable bounds to a ground value and `bound` for a variable bounds to any entity (value or variable). This distinction allows predicates to wake up on aliased variables. For example, with the constraint $X \neq Y$, we wish to detect if $X = Y$ even if none of these variables is bound. Overall, thanks to this flexible event system and the close integration with Herbrand terms, new constraint systems can be added as libraries into Eclipse without the need to modify the intrinsics of the system.

Among the popular CLP systems, we also find SICStus Prolog [CM12] and SWI-Prolog [WSTL12]. Both are very complete Prolog environments. SICStus Prolog has one of the most complete implementations of CLP's domains among all systems. An extensive comparison and correctness study of various modern CLP systems is available in [Tri14].

To conclude this part, we must mention [HK12] that proposes a concise, yet efficient, SAT and SMT solver written entirely in Prolog. In particular, they show how to program the watched literals optimization consisting in moving propagator triggers during the execution. It demonstrates that the suspension mechanism can be used to program complex and dynamic event-based algorithms while retaining good efficiency.

2.5.3 Lattice-based semantics of CLP

We investigate the semantics of CLP based on the lattice hierarchy in the same way as for CCP. A major difference is that CLP supports both "don't care" and "don't know" nondeterminism. In comparison with more traditional semantics, our formulation has the advantage of making explicit both forms of non-determinism. We believe it reduces a gap between language theory and implementation.

Since we have two levels of nondeterminism, there are two queueing strategies for navigating between the three lattice layers. As a first high-level notation, we represent the CLP semantics with a diagram in Figure 2.5. The nodes are the lattice structures and the edges are labelled with the semantics rules and are used to move from one lattice to another. An important additional point is that a rule applied on a lattice L_i has access to any available lattice L_j where $j \geq i$. For example, the branch and bound function (Definition 1.49) is called on every node of the search tree but transforms the search tree as a whole, and thus it belongs to the upper lattice layer. We present the lattice structures involved in the semantics

of CLP, and then explain the different rules.

CLP lattice hierarchy

In contrast to CCP, the state of a program must be part of the lattice hierarchy since, on backtrack, the set of clauses must be restored. In this respect, the program must also be part of the lattice structure. We represent the state of a logic program by a multiset of clauses' heads such that the heads of disjunctive clauses are distinguished by an integer. This is made clear in the following example.

Example 2.10 (State representation of Horn clauses)**.** Consider the following logic program:

```
length([], N) :- N is 0.
length([H|L], N) :- length(L, N1), N is N1 + 1.
```

The states encoding these clauses are $\texttt{length}_1$ and $\texttt{length}_2$ (numbered from top to bottom). The trace of the program `length([a,b], N)` is:

$$\{\texttt{length}_2\}$$
$$\{\texttt{length}_2, \texttt{length}_2\}$$
$$\{\texttt{length}_2, \texttt{length}_2, \texttt{length}_1\}$$

⌟

Based on this representation, we can now define the code state of a program.

Definition 2.11 (Code state of a program)**.** *The set of all predicate symbols is denoted by $Pred$. The code state of a program is composed of the set of predicates already unfolded, denoted by T, and the set of active predicates not yet unfolded, denoted by A. These sets are defined as:*

$$T = \mathit{Store}(L_2^{\scriptscriptstyle CLP} \times \mathbb{N})$$
$$A = \mathit{Store}(L_2^{\scriptscriptstyle CLP})$$

where an element $t \in T$ is the trace of the program and an element $(p, i) \in t$ is a predicate p with its clause number i. The active predicates in A are not yet numbered since they have not been unfolded yet. The lattice of the code state of a program is defined as follows:

$$\mathit{Code} = \langle T \times A, (t, a) \models (t', a') \text{ if } t \models_T t' \wedge ms(\pi_1'(t)) \cup ms(a)) \models_{ms} ms(a') \rangle$$

where ms is a function turning a store into a multiset, and the order $\models_{ms}$ is the superset inclusion relation $\supseteq$. The order indicates that the trace of t' must be contained inside the one of t and that the active predicate in a' must either be inactive or active in (t', a'). Furthermore, we transform the store into a multiset since the store order of the active predicates is not relevant yet—they have not been selected yet.

Unfortunately, the trace of the program must also be saved in the structure. The reason is that otherwise the lattice could not represent diverging logic programs such as `p :- p`. Without the trace multiset, such programs could not be represented by a monotonic function over L_3^{CLP}. Tabling extension to logic programming is a paradigm in which redundant computations are avoided. For example, XSB is a tabled logic programming language that also supports tabling with CLP [SW12]. However, we leave this study for future work.

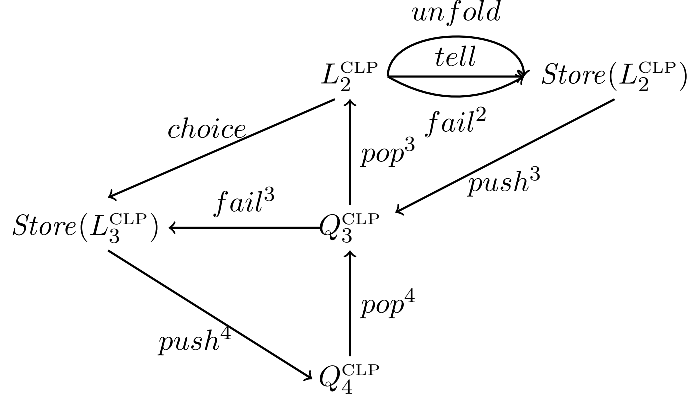


Figure 2.5: CLP exploration strategy.

Back to CLP structures, the code state lattice of a program can be combined with the lattice of CSPs L_3. It forms a new hierarchy that is proper to CLP.

Definition 2.12 (CLP hierarchy)**.** *Given the lattice L_3, the lattice hierarchy of the CLP semantics is defined as follows:*

$$\begin{aligned} L_2^{CLP} &= \{\top\} \oplus Pred \oplus \{\bot\} \\ L_3^{CLP} &= L_3 \times Code \\ L_4^{CLP} &= \mathscr{A}(L_3^{CLP}) \end{aligned}$$

with orders inherited from the corresponding derivations.

The predicates are stored in a lattice that is paired with the CSP, it is defined at the second level but actually, we have $L_2^{\text{CLP}} = L_1^{\text{CLP}} = L_0^{\text{CLP}}$. The two queueing strategies are defined over L_4^{CLP} and the store of active predicates A in L_3^{CLP}. The queueing strategy is given over A with the pair of functions $(pop_3, push_3)$ and the one over L_4^{CLP} with the pair of functions $(pop_4, push_4)$. The first one is used to select the next goal to instantiate while the second one is used to select the next clause to search in case of disjunctive predicates. We write Q_4^{CLP} for the queuing derivation of L_4^{CLP}, and respectively Q_3^{CLP} for the one of L_3^{CLP}.

Multi-domains

An important aspect of the lattice framework is to derive new constraint systems by composition of existing ones. This is particularly interesting for CLP since it

merges the domain of Herbrand terms and another one, for example the intervals over a finite subset of $\mathbb{N}$. In this case, the lattice L_1 of the (constraint) lattice hierarchy can be defined as:

$$L_1 = \{\top\} \oplus (\mathscr{I}(\mathbb{N}) \,\dot\cup\, H) \oplus \{\bot\}$$

where H is the set of Herbrand terms that can be build out of a given signature. The disjoint union $\dot\cup$ (Definition 1.19) builds a new lattice which is the "type-erasure" of two domains. Of course, it does not mean that constraints can be freely defined over both domains, such a verification is left to the function $[\![c]\!]$ translating a constraint c into a propagator.

Cooperation between multiple domains is a research field in its own respect and it is extensively studied for combining SMT theories (see Section 2.2.4). A research axis is the study of their compositions in the context of the lattice theory. We believe it would be useful for better combining non-convex theories using techniques from the field of constraint programming.

CLP semantics rules

$$\text{fail}^2\,\frac{\{\} = \mathit{instantiate}(\langle d, P\rangle, g(t_1, \ldots, t_n), D)}{D \vdash (q, \langle d, P\rangle, G, g(t_1, \ldots, t_n)) \to (q, \top, G, \bot)}$$

$$\text{unfold}\,\frac{\{(i, (\langle d', P'\rangle, B))\} = \mathit{instantiate}(\langle d, P\rangle, g(t_1, \ldots, t_n), D)}{D \vdash (q, \langle d, P\rangle, G, g(t_1, \ldots, t_n)) \to (q, \langle d', P'\rangle, G \sqcup \{g_i\}, B)}$$

$$\text{choice}\,\frac{\begin{array}{c} C = \mathit{instantiate}(\langle d, P\rangle, g(t_1, \ldots, t_n), D) \qquad |C| > 1 \\ C' = \{(i, (\langle d', P'\rangle, (\pi'_1(G) \cup \{g_i\}, \pi'_2(G) \cup B))) \mid (i, (\langle d', P'\rangle, B)) \in C\} \end{array}}{D \vdash (q, \langle d, P\rangle, G, g(t_1, \ldots, t_n)) \to (q, C')}$$

$$\text{tell}\,\frac{\langle d, P \cup \{[\![c]\!]\}\rangle \Rightarrow \langle d', P'\rangle}{D \vdash (q, \langle d, P\rangle, G, c) \to (q, \langle d', P'\rangle, G, \bot)}$$

$$\text{fail}^3\,\frac{\mathit{ssd}(\langle d, P\rangle) = \mathit{false}}{D \vdash (q, \langle d, P\rangle, G) \to (q, \bot)}$$

Figure 2.6: Lattice-based semantics rules of CLP.

We detail in Figure 2.6 the semantics rules mapping a lattice to another as shown in the former diagram. To start with, we consider the instantiation of a

goal given by the rules *unfold* and *choice*. The choice of one rule or the other is driven by the determinacy analysis: if the goal is determinate then we use *unfold* and otherwise we use *choice*. Since instantiation is redundant in both rules, we extract it in the following definition.

Definition 2.13 (Goal instantiation)**.** *Let $g(t_1, \ldots, t_n)$ be the goal to instantiate where $t_1, \ldots, t_n$ are terms and D the predicate definitions forming the logic program. Let a possible predicate $D_i \in D$ where D_i has the form $p_i(u_1, \ldots, u_m) : -B$ such that it matches the goal head $p_i = g$ and has an identical arity $n = m$. We note V_i the set of free variables in D_i. We define a function that instantiates a goal to a possible predicate:*

$$\begin{array}{l} instantiate_one(\langle d, P\rangle, g(t_1, \ldots, t_n), D_i) \mapsto \\ \quad \begin{cases} \{(\langle d', P'\rangle, B')\} & \text{if } ssd(\langle d', P'\rangle) \neq false \\ \{\} & \text{otherwise} \end{cases} \end{array}$$

where the instantiated predicate is computed by the following algorithm:

$$\begin{array}{ll} M = \{(d_i, l_i) = alloc(d_{i-1}, \bot) \mid v_i \in V_i\} \text{ with } d_0 = d & \text{(allocate free variables)} \\ D'_i = D_i[v_1 \to l_1, \ldots, v_n \to l_n] & \text{(substitute variables for locations)} \\ D'_i = p'_i(u'_1, \ldots, u'_n) : -B' & \\ \langle \bigsqcup \pi'_1(M), P \cup \{[\![t_1 = u'_1]\!], \ldots, [\![t_n = u'_n]\!]\}\rangle \Rightarrow \langle d', P'\rangle & \text{(unification)} \end{array}$$

Given the sequence $AP = \langle D_1, \ldots, D_n\rangle \subset D$ of possible predicate definitions, we define

$$\begin{array}{l} instantiate(\langle d, P\rangle, g(t_1, \ldots, t_n), D) \mapsto \\ \quad \{(i, instantiate_one(\langle d, P\rangle, g(t_1, \ldots, t_n), D_i)) \mid D_i \in AP\} \end{array}$$

which maps to the (possibly empty) set of applicable predicates.

The domain and codomain of the transitions change according to the layer of the lattice—this is why the diagram in Figure 2.5 is particularly useful. The semantics rules, as shown in Figure 2.6, are mostly built around the instantiation of a predicate. In particular, according to the cardinality of the set returned by *instantiate*, we have three cases:

- If it is empty, then the rule $fail^2$ is invoked and it sets the CSP to $\top$ since it is unsatisfiable. Then, it is detected by the rule $fail^3$ that triggers backtracking in Q_4^{CLP}.
- If it is a singleton, the rule *unfold* creates a store of new goals B that will be pushed onto the queue of active predicate in Q_3^{CLP}.
- In case multiple predicates are applicable, the rule *choice* jumps from L_2^{CLP} to Q_4^{CLP} by creating a choice point for each possible alternative.

In addition, the rule *tell* is an atomic tell operation: it adds a constraint and performs the propagation immediately. In contrast with *unfold*, it does not generate more active predicate and only impacts the running CSP. For example, this is the strategy used in GNU Prolog. If the CSP becomes unsatisfiable, it is detected by the rule $fail^3$.

Discussion

In comparison with existing CLP semantics, notably the work in [MJMS98], the hierarchical approach fully acknowledges the presence of nondeterminism in its structures. This is in contrast with semantics specialized for left-to-right literals selection and depth-first search, for which the queueing mechanism is left implicit in the rules. In this respect, it is an advantage to articulate the semantics around two abstract, but explicit, queueing strategies.

We hope that this framework can be the starting point for a more in-depth analysis of the logic paradigm. An axis to develop in future works is the comparison between the various determinacy analysis—the one described here being the most basic. Furthermore, we should be able to formalize extra-logical operators within the framework. For example, the cut operator can be specified with a semantics rule using the delta operator Δ_4. A challenge would be to keep track of the subtree that needs to be pruned by this operator.

2.6 Search languages: L_4 and above $(\sqcup, \vDash, \Delta)$

In 1979, Kowalski writes the paper "Algorithm = Logic + Control" [Kow79] in which he elaborates that logic specification should be kept separated from the way it is solved. It was very influential in the development of the logic paradigm and was largely realized by keeping the control part internal to the language's implementation. However, over the years, more and more combinatorial problems have demonstrated that control was essential for efficient solving, but at the same time, that a generic control strategy could not be efficient for every problem. From these initial considerations, control primitives have slowly emerged, without being fully acknowledged as a part of the specification. It led to the development of search languages and frameworks that aim at capturing the essence of control. Before going through the various proposals, it is a good time to step back and ask: what is control?

As a first tentative, we can say that control is any function participating in the decision of how the computation evolves. From the point of view of the lattice hierarchy, a logic specification is an element U of any lattice L_i. The control specification is a function mapping this element to an element S, if any, in the solution space of L_i such that $S \models_i U$. To do so, the control part is defined over a lattice L_j where $j \geq i$. Note that logic specification can be defined over any lattice, and not only L_3. For example, consider the statement: "find the best solution".

We saw in Section 1.6 that such optimization problems are defined with a function over L_4. From this example, we realize that the frontier between logic and control is blurry since the logic specification directly impacts the control strategy.

Therefore, rather than focussing on the distinction between logic and control, it is more appropriate to focus on why we want to keep them separated. The major goals of separating logic and control is to be able to try and test several control strategies on an identical logic model, and oppositely to use an identical control strategy on several distinct models. The first is mandatory to empirically evaluate a strategy against a model, and the second to provide strategies reusable across different models. The challenge is to design useful languages' operators to program such strategies. In particular, the strategy language should be compositional: given two strategies A and B, how can we obtain a third strategy C by composition of the two first.

Language proposals for programming search strategies continuously appeared for the last two decades, and there is no proposal widely accepted yet. We do not give an exhaustive review of every search language but instead focus on five issues in search languages and the attempts to solve them.

- First issue: Two search strategies cannot be composed.
- Second issue: Impossibility to arbitrarily prune branches.
- Third issue: Non-backtrackable variables are lacking.
- Fourth issue: Control operators are duplicated for different layers of the hierarchy.
- Fifth issue: Two search strategies cannot freely communicate.

We organise this section such that these issues are unfolded as we survey the different search languages. We start to survey the extensions to CLP for programming search strategies (Section 2.6.1). It is followed by various attempts to incorporate search capabilities into different (non-logical) paradigms such as imperative or functional (Section 2.6.2). Finally, we survey "pure search languages" proposing various control operators for exploring a state space and for combining existing strategies (Section 2.6.3).

2.6.1 Search in logic languages

Search primitives are part of most CLP systems as built-in predicates that users can assemble in order to obtain a search strategy. The most basic built-ins are the various `labelling` predicates for implementing a customized branching strategy, available in virtually every CLP language. As a starting point to exemplify the support of search in logic languages, we consider Eclipse which has one of the most extensive built-in supports for search [AW07]. Eclipse is equipped with two main search predicates: `search/6` and `bb_min/3` where

```
split_mid_bd(X, _) :- integer(X), !.
split_mid_bd(X, 0) :-
  middle(X, M), X #=< M, !,
  split_mid_bd(X, 0).
split_mid_bd(X, Dis) :-
  middle(X, M),
  (
    NDis is Dis - 1,
    X #> M,
    split_mid_bd(X, NDis)
  ;
    X #=< M,
    split_mid_bd(X, Dis)
  ).
```

(a) CLP

```
split_mid_bd(X, Dis) :-
  tor_merge(bd(Dis), split_mid(X)).

split_mid(X) :- integer(X), !.
split_mid(X) :-
  middle(X, M),
  (
    X #> M
  tor
    X #=< M
  ),
  split_mid(X).
bd(Dis) :-
  (
    Dis > 0,
    NDis is Dis - 1,
    bd(NDis)
  tor
    bd(Dis)
  ).
```

(b) Tor

Figure 2.7: Left-branch bounded discrepancies with Prolog and Tor.

- `search(Vars, Proj, VarOrder, ValOrder, Method, Options)` is branching over the variables `Vars` with a variable and value ordering specified by the predicates `VarOrder` and `ValOrder`. In case `Vars` contains terms, the integer variable `Proj` indicates that the *Proj*th sub-term is a domain variable. More interestingly, `Method` is one of about 10 different search strategies such as limited-discrepancy search (LDS) and pruning strategies such as dynamic symmetry breaking. Finally, `Options` is a list of options to retrieve the number of backtracks or to bound the exploration to a limit of nodes.
- `bb_min(Search, Cost, Options)` executes the strategy `Search` with a branch and bound (BAB) algorithm minimizing the variable `Cost`. Various options can be specified such as changing the underlying BAB strategy and monitoring the progress of the search.

Although Eclipse provides a large numbers of base primitives, it is hard to program new strategies without transforming the existing framework. For example, limited-discrepancy search (LDS) with a bounded depth cannot be generically obtained through the existing infrastructure of `search/6`.

There are even more fundamental problems in Prolog that make any extension hard to be fully satisfactory. As an example, we take an excerpt of the documen-

tation of Eclipse[12] commenting on LDS[13]:

> The original LDS paper stated that the discrepancy to be tested first should be at the top of the tree. Our implementation tries the first discrepancy at the bottom of the tree. [...] This change is imposed by the evaluation strategy [of Prolog] used and can not be easily modified.

This is because children nodes are pushed from left to right onto the queue when evaluated in Prolog. Actually, we can solve this problem by manually reversing the order of the branches and considering left instead of right discrepancies in the implementation of LDS. However, this fix is not compositional: we reversed the branches in LDS but another strategy could need them in the left to right order.

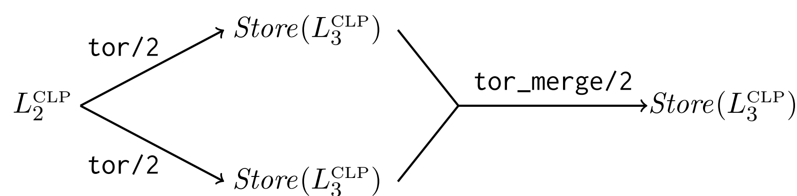


Figure 2.8: TOR conceptual diagram.

To make our discourse more concrete and as a basis for what follows, we show this variant of discrepancy-bounded search in Figure 2.7a with the CLP system GNU Prolog. The predicate `split_mid_bd(X, Dis)` takes a variable X and explores its tree by taking at most Dis left branches. For clarity, this example simplifies the problem by splitting on a unique variable X and by stopping after Dis discrepancies instead of restarting with an increased bound.

First issue: Two search strategies cannot be composed

A problem with `split_mid_bd` is that the branching and the discrepancy-bounded strategies are entangled. If we want to program a depth-bounded strategy with the same branching function, we must program the composition of these two by hand and duplicate parts of the code.

To overcome this issue of non-compositionality, Schrijvers et al. [SDTD14] designed the `tor` predicate. In essence, it proposes to exchange the traditional disjunctive Prolog predicate `;/2` with a novel `tor/2` predicate. This predicate serves two purposes: to create and combine search trees. The search tree described by two predicates can be merged with the predicate `tor_merge/2` as long as they both have a unique `tor` synchronization point. This is exemplified in Figure 2.7b with a discrepancy-bounded strategy similar to the former one in CLP. The interesting aspect of `tor/2` is that both strategies `split_mid` and `bd` are specified independently.

[12] `http://eclipseclp.org/doc/bips/lib/fd_search/search-6.html`, accessed on 3rd February 2018.
[13] See Example 1.11 for an explanation of LDS.

They are next assembled using `tor_merge/2` to form the full search strategy. To its full extent, this extension allows us to have a collection of strategies that can be assembled but also extended by users.

The traditional Prolog disjunctive predicate `;/2` pushes alternatives onto an (implicit) queue, and thus can be seen as the join operator $\sqcup_4$. In addition, `tor/2` is creating synchronization points that can be merged *via* `tor_merge/2`. Its lattice-based semantics can be described as the join operator over $Store(L_3^{\text{CLP}})$. To illustrate this, we draw a diagram in Figure 2.8 showing how two `tor/2` predicates are executed and then merged. The control flow forks when we use two predicates `tor`, and then their results are joined with `tor_merge` before the nodes created by `tor` are pushed onto the queue. Hence, the conceptual idea underlying `tor` is to concurrently create two search trees, and to arbitrate their final combination with `tor_merge`. In practice, only one predicate is active at any time: the first `tor` executed temporarily stores its branches in a global branch store (of type $Store(L_3^{\text{CLP}})$). Whenever the next predicate `tor` is encountered, it composes its left and right branches with the ones already present in the store.

Some of the same authors acknowledge in [SWDD14] that the tor extension lacks a proper semantics. They argue it is hard to work out new strategies without being accustomed to the details of the implementation. Therefore, they propose a better semantics of `tor`, renamed the *entwine* operator, using functional monads with techniques reminiscent of monadic constraint programming (MCP) [SSW09]. Whilst being better defined, the search strategies described are harder to read than the `tor` approach, mainly due to the heavy background required in functional language theory.

Second issue: Impossibility to arbitrarily prune branches

Looking again at the two versions of the discrepancy-bounded strategy in Figure 2.7, we realize there is a slight operational difference. The CLP version stops taking left branches once Dis is equal to 0. In the `tor` version, even if we start with a discrepancy of 0, it pushes exactly $2d$ nodes onto the queue for a tree of depth d. Despite the fact we know a left node with a discrepancy of 0 is failed before it is pushed, there is no direct way to indicate that we want to discard it. This is because the creation of the search tree is limited to 0—if we force to backtrack—or 2 children nodes. Actually, what is really lacking is the support for a delta operator over $Store(L_3^{\text{CLP}})$ in order to discard a specific node. This example shows that, although we have control operators over L_4, and more specifically over $Store(L_3^{\text{CLP}})$, we still lack of an abstraction to precisely manipulate the construction of the search tree.

Third issue: Non-backtrackable variables are lacking

A more general problem with logic languages is that every variable is always backtracked, even when we do not want to. It is a blatant issue when programming

```
var 0..4: x;

middle(X, M) :-
  M = (min(X) + max(X)) div 2.
split_mid(X, XCard) :-
  XCard > 0,
  middle(X, M),
  (
    X <= M
  ;
    X > M
  ),
  split_mid(X, XCard - 1).
split_mid(X, 0).

:- split_mid(x, 2).
output [show(x)];
```

(a) Model in ClpZinc

```
var 0..4: x;
var 0..4: X3;  var 0..1: X5;  var 0..4: X4;
var 0..1: X6;  var 0..4: X2;  var 0..4: X1;
constraint X5 = 0 <-> x <= (X1 + X2) div 2;
constraint X6 = 0 <-> x <= (X3 + X4) div 2;
solve :: seq_search([
    indexical_min(X1, x),
    indexical_max(X2, x),
    int_search([X5], input_order,
      indomain_min, complete),
    indexical_min(X3, x),
    indexical_max(X4, x),
    int_search([X6], input_order,
      indomain_min, complete)
  ]) satisfy;
output [show(x)];
```

(b) Model unfolded in MiniZinc

Figure 2.9: Split in the middle branching strategy using ClpZinc.

search strategies. A primer example is the node-bounded search strategy which stops after exploring a fixed number of nodes. Clearly, we do not want this variable to be backtracked, and the only solution in existing system is to rely on mutable and non-backtrackable variables extension of logic systems. The problem with global variable is that it breaks the monotonicity property of a Prolog computation. Hence, it is hard to reason about and does not fit well in the semantics of Prolog. Moreover, such extensions are not portable across different logic languages and such variables must be managed manually by the programmer. However, we require non-backtrackable variables at countless places in the implementation of search strategies as shown in [SSW09, SDTD14].

We can give a solution with our hierarchical view of computation. For example, given a node count variable defined over $LMax(\mathbb{N})$, in logic language this variable is added into L_2 whereas we would need to add this variable into Q_4 with a new lattice $Q'_4 = Q_4 \times LMax(\mathbb{N})$. This node counter must be at the same level than the queue of nodes since it reflects a property of Q_4. More generally, a solution is to allow the user to define variable at any level of the lattice hierarchy in order to prevent backtracking of their values.

Another approach: L_4 collapsing into L_3

ClpZinc is a radically different approach that uses constraints in the lattice L_3^{CLP} to program a search strategy [MFS15]. To accomplish this, it relies on reified constraints which are higher-order constraints reifying the domain entailment of a constraint c into a boolean variable b. For example, reifying the constraint $x > 2$

is done as:

$$b \Longleftrightarrow x > 2 \mapsto \begin{cases} b = 0 \Rightarrow x \leq 2 \\ b = 1 \Rightarrow x > 2 \\ x > 2 \Rightarrow b = 1 \\ x \leq 2 \Rightarrow b = 0 \end{cases}$$

In essence, reified constraints are a form of conditional statement: if b is set to 1 then $x > 2$ is enforced, and otherwise $x \leq 2$ is enforced (and dually if $x > 2$ is entailed or disentailed first). The idea conveyed in [MFS15] is to use such conditionals to fully describe the search tree of a CSP before it is even solved. Basically, they associate a boolean variable b to each node of the search tree, when it is true they activate the left branch of the node and otherwise they activate the right branch. The constraint reified through b is the one used in branching: either c or $\neg c$ will label the branch.

Concretely, ClpZinc extends the modelling language MiniZinc with Horn clauses used to describe the search tree. We consider an example in Figure 2.9 that splits in the middle of a variable x with the domain [0..4]. In Figure 2.9a, the MiniZinc model is accompanied by a branching predicate `split_mid/2` similar to a pure CLP predicate. This predicate is evaluated at compile-time by the ClpZinc compiler in order to generate the static search tree with reified constraints. Therefore, we must know the depth of the search tree before even executing it; this is the role of `XCard` which is a precomputed maximum tree depth bound. In our case, since the unique variable `x` takes a value in [0..4], with the splitting strategy, the tree has a depth of 2 at maximum. Importantly, we need `XCard` because we are not evaluating the constraints but just generating them, and thus we can not rely on the failure of the constraint store to stop the branching predicate.

The generated model is shown in Figure 2.9b. The variables `X5` and `X6` are the boolean variables of the two successive decisions that we must take in order to explore the full tree. A basic search strategy will enumerate the boolean variables to successively activate every branch of the tree. They show in [MFS15] that more complex search strategies such as dynamic symmetry breaking can be statically generated.

This approach is interesting because it applies propagation algorithms to the tree exploration and further investigations could lead to novel propagation algorithms or unforeseen search strategies. However, partial evaluation is a double-edged sword because the size of the generated program depends on the number of variables and the size of the domains. Finally, it is orthogonal to all of the three issues mentioned above, but unfortunately it does not help to design more compositional search strategy.

2.6.2 Search in other paradigms

Beyond search in the logic paradigm, there are a number of languages that integrate operators for controlling and using backtracking. In particular, we review some

proposals in the imperative and functional paradigms.

Imperative search language

ICON is an imperative language designed in 1977 particularly suited for processing text and symbolic data [GG93, GG97]. It is one of the first imperative languages to integrate notions of success and failure of the computation directly into its semantics. Furthermore, it also provides automatic backtracking in case of failure through the notion of generators. A generator is a function that generates a new element if the former did not lead to a solution.

An interesting concept in ICON is to separate data and control backtracking. Data backtracking restores the data to a former value while control backtracking restores the control flow only. Usually, as in Prolog, backtracking the control flow automatically backtracks the associated data. In ICON, some operations on data (such as the assignment) can be persistent across backtracking. To a limited extent, this language provides a solution to the third issue on mutable non-backtrackable variables. Finally, note that this language is still available and maintained forty years after its creation[14].

In 1997, the Alma-0 language proposed to integrate search into Modula-2 [ABPS98] using concepts from logic programming. It provides a choice point operator to implement branching and a commit operator to implement pruning similarly to the cut in Prolog. Moreover, in [AS99], they integrate constraints using an atomic tell operator similar to CCP, but they do not provide support for the entailment operation.

Overall, constraints mixed with the imperative paradigm—including modelling languages (Section 2.3.2)—encounter a limited success. A possible explanation is the development of constraint libraries within the same paradigms that are more efficient.

Monadic constraint programming

Monadic constraint programming (MCP) is one of the most sophisticated libraries for programming search strategies [SSW09]. Although it is a library programmed in Haskell, and not a language per se, MCP is instrumental in the design of recent search languages such as search combinators [STW$^+$13]—described below. It is also a strong proposal to solve the three first issues presented in Section 2.6.1. To give a brief overview of MCP, we show two of its most important data structures in Figure 2.10. The search tree is explicitly represented with `Tree solver a` where:

- `solver` provides an interface to interact with an underlying solver, and
- `a` is the type of the expected result of the computation.

[14] See the website `https://www2.cs.arizona.edu/icon/`.

```
data Tree solver a where
  -- Operations on the constraint solver
  Return a
  NewVar (Term solver -> Tree solver a)
  Add (Constraint solver) (Tree solver a)
  -- Search support
  Try (Tree solver a) (Tree solver a)
  Fail
  Dynamic (solver (Tree solver a))
```

(a) Tree monad

```
class Transformer t where
  type EvalState t :: *
  type TreeState t :: *
  initT ::
    t -> (EvalState t, TreeState t)
  leftT, rightT ::
    t -> TreeState t -> TreeState t
  nextT :: SearchSig solver q t a
  -- Default implementation
  leftT _ = id
  rightT = leftT
  nextT = eval
```

(b) Transformer structure

Figure 2.10: Monadic constraint programming: excerpt of data structures.

The constructors of `Tree` denote actions that will alter the `solver` (L_3 structure) and the queue (L_4 structure) when executed. The mapping of the variants with lattices is as follows: `NewVar` corresponds to $\sqcup_2$, `Add` to $\sqcup_3$, `Try` to $\sqcup_4$ and `Fail` to Δ over $\mathit{Store}(L_3)$. Using the variant `Dynamic`, the tree can be lazily expanded using information contextual to the current state of the solver. Finally, this tree structure is evaluated by a function `eval` modifying the solver and queue structures accordingly.

The evaluation function explores exactly the search tree induced by the tree structure. On top of it, we can program *search transformers* that dynamically build and alter the search tree during the exploration. We must implement the `Transformer t` interface shown in Figure 2.10b. The central idea is to leave the function `eval` triggering transformers' hooks when we start the search (`initT`), explore the left or right child (`leftT` and `rightT`) and continue the exploration with any next node (`nextT`). Decisions taken inside the transformer are made using two variables of type `EvalState` and `TreeState`, respectively for storing information global to the search tree or local to the current path in the tree. The functions `leftT` and `rightT` trigger modifications on the tree state whilst `nextT` triggers the exploration of the remaining search tree. The latter function injects new constraints and prunes nodes in the search tree.

Example 2.11 (Depth-bounded search transformer)**.** To illustrate this behavior, we consider the depth-bounded search transformer:

```
newtype DepthBoundedST = DBST Int

instance Transformer DepthBoundedST where
  type EvalState DepthBoundedST = Bool
  type TreeState DepthBoundedST = Int
  initT (DBST n) = (False,n)
```

```
leftT _ ts = ts - 1
nextT tree q t es ts
| ts == 0 = continue q t True
| otherwise = eval tree q t es ts
```

`DepthBoundedST` acts as a proxy cutting the tree beyond some predefined depth. The evaluation state contains a boolean indicating if it already pruned the tree or not, and the tree state contains the current depth. Each time we take a left or right branch (note the default implementation in 2.10b), we decrease the current depth. Once we hit a depth of 0, the function `nextT` is using `continue` to bypass the rest of the tree evaluation by extracting a new node from the queue. Otherwise we continue the evaluation of the current node and subtree, and the transformer has no impact on the search. ⌟

Finally, they achieve composition by stacking up search transformers where one has the responsibility to call the next transformer. Transformers are applied one after another on each node of the search tree using a continuation-passing style scheme: each transformer calls a continuation executing the transformer just below or directly jumps to the next node (enabling pruning).

Interestingly, MCP partially solves all the first three issues presented in Section 2.6.1. The monadic approach and search transformers enable the user to compose search strategies (first issue) and to prune arbitrary branches (second issue). The third issue is tackled by `EvalState` that provides a clean way to compose variables with L_4 and `TreeState` with L_3. However, we still lack the delta operator over L_4 to program, for example, backjumping.

Beyond this short introduction, MCP is also generic with regard to the queue, branching and restoration strategies. MCP elegantly captures and separates the numerous concepts of constraint solving. However, understanding how these concepts interact is hard, but it is a mandatory task for the users who want to program new strategies. We believe that overcoming such limitations is the role of programming languages, by designing high-level search operators and semantics that cleanly express the conceptual ideas behind search strategies. We must also credit MCP as an important resource for the design of the lattice-based framework of the first chapter.

Search in multi-paradigm languages

`Oz` is a multi-paradigm language with support for constraint and search [RH04]. We already mentioned `Oz` in Section 2.5.1 as the successor of `AKL` and proposing nested computation space. The outcome of a computation space (where nondeterministic computations are encapsulated) can be manipulated as a first-class entity whereas it is built into the semantics of Prolog. Formally, a computation space is a function from L_3 to $Store(L_3)$ where the emptiness of the resulting store is interpreted as a failure. Complex search strategies can be programmed such as branch and bound, but also restoration strategies such as recomputation [Sch02].

Moreover, computation spaces have been used to program higher-order constraints such as the negation of any constraint and reified constraints. However, there is no specific support for the composition of the strategies which remains a low-level task. Finally, we note that computation space has been instrumental in the design and understanding of constraint solving libraries such as GeCode [STL14].

Computation spaces are also explored in the context of the functional logic language Curry where they introduce the `try` function [HS98]. Similarly to Oz, this function maps a CSP in L_3 to a store of branches $Store(L_3)$. An original feature of this work is to use the lazy evaluation of Curry to elegantly deal with infinite search tree.

2.6.3 Control operators for search

Designing a language to program the search is not an easy task, especially when taking into account compositionality issues. Research in this field has been overloaded with two different considerations: integrating search inside an existing paradigm and finding the right set of search operators. Since it is very challenging to try to solve both problems at once, approaches based on combinators emerged. A combinator is essentially a higher-order function that combines other functions. Based on this idea, language designers can exclusively focus on the search combinators, and the possible interactions with a host language are limited to a mere interface. Moreover, it has the advantage of simplifying the overall design and to obtain concise and explicit search strategies.

Early constraint search languages appeared around 1998 with Localizer [MVH99], Salsa [LC98] and OPL [VHM99]. More recent approaches include Comet [VHM05] (successor of Localizer), search combinators [STW+13] and MiniSearch [RGST15]. The control primitives offered by these languages are substantially overlapping, thus we limit our study to a few central proposals. In particular, languages specialized to local search such as Localizer and Comet are left out of scope.

Fourth issue: Control operators are duplicated for different layers of the hierarchy

In Section 2.6.1, we discussed the third issue about variables global to the search tree that should not be backtracked. The proposed solution is to explicitly situate a variable in a layer of the hierarchy. In most search languages, they tackle this issue either by relying on the host language for global variables such as in search combinators [STW+13, RGST15] and Salsa [LC98] or by duplicating operations over global and backtrackable variables such as in OPL [VHM99].

The first solution relies on the underlying solver to access statistics such as the depth of the tree, number of right branches and nodes explored. Therefore, beyond posting constraint in L_3, the treatment of data is left to a host language. It is strange that a search language, designed to operate over a tree, is unable to

express simple tree's properties. An advantage of search combinators and Salsa is to have a limited set of combinators.

As for the second solution in OPL, they duplicate control primitives according to the layer of definition of the variable. The assignment has two versions: `<-` and `:=` depending on whether it is backtrackable or not. The statements `while` and `if` work on global and grounded variables while the statements `when`, `onValue`, `onRange` and `onDomain` are statements reacting to variables from L_0 to L_3. These latest statements are reminiscent of indexicals and concurrent logic programming where a process can suspend and wait on some conditions. They show in [VHPP00] that `<-` and events conditions allow users to program dynamic branching strategy in a scheduling problem.

One of the reason to duplicate statements is that conditions expressed in L_3 can be *unknown* while conditions over global variables are usually equal to *true* or $false$. However, the duplicated statements make OPL one of the heaviest languages in terms of number of concepts and operators.

Control operators for L_4 and above

OPL and Salsa provide extensive support for programming the branching strategy, and search combinators mostly rely on the host language. In particular, many operators are defined to program the value and variable ordering (Definition 1.46) according to various criterion. The branching operators manipulate the store of nodes $Store(L_3)$ before the nodes are pushed onto the queue. Accordingly, they also propose pruning operators over this structure: Salsa's s `where` f prunes the current subtree if the function f returns true, and `prune` does a similar job in search combinators. Salsa proposes a pruning operator over L_4 by stopping the search under various conditions with s `until` f that stops the strategy s when f returns true. OPL furnishes a similar operator with `applyLimit` id s.

Interestingly, OPL is one of the only languages to allow the user to program its own queuing strategy. Basically, the operator `applyStrategy` id s executes a strategy s with the queueing strategy id which is of the form:

```
SearchStrategy id (args) {
  evaluated to e1;
  postponed when e2;
}
```

where e_1 is a function evaluating the current node and e_2 describes on what condition we switch to another node. This way, best-first search can be implemented with customized expression.

We terminate with some examples of combinators to program the search in L_5 which is especially useful for restart-based search strategies. In search combinators, `restart(`e, s`)` restarts a strategy s whenever a condition e become $true$. Similarly, `or` and `portfolio` restart the search respectively until a solution is found and until we explore the full tree. OPL provides `solve(`s`)` that performs nested search: s is

executed with its own queue of nodes. The variants `minof(`e`, `s`)` and `maxof(`e`, `s`)` are specialized for retrieving the best value of the alternatives in the nested search. It is useful to implement shaving (or probing) techniques that collect information on all children nodes before committing to a subtree. All operators are actually creating new search trees and thus can be viewed as the join operator $\sqcup_5$.

Fifth issue: Two search strategies cannot freely communicate

A consequence of the fourth issue is the difficulty to communicate data among search strategies. Search strategies communicate indirectly through internal and hidden data to decide what strategy to execute next, for instance exhaustiveness with portfolio in search combinators and failure of branching in Salsa. A similar approach is employed in MCP by stacking up search transformers. The search transformers are layered such that a transformer decides to execute or not its child. The communication is unidirectional in the sense that a child cannot easily communicate information to its parent. In Tor, different search strategies communicate on mutable global variables to indicate if a pruning occurred [SDTD14]. In this case, we fall back in the traditional problematic of concurrent read and write in imperative variables. Solutions have already been explored in the context of concurrent logic programming where predicates communicate concurrently, but we are not aware of similar proposals for search strategies.

2.7 Conclusion and discussion

Overall, the two predominant layers tackled by language abstractions are L_3 and L_4. Languages supporting only L_3 usually embed the notion of infinite computation which becomes problematic when coupled to backtracking. In this respect, the constraint logic paradigm is the most advanced proposal to merge L_3 and L_4 into a single language. This leads to several complications, notably compositionality of search strategies, and this is why a bunch of proposals are specialized for L_4 uniquely.

This chapter surveyed some of the most well-known languages in the field of constraint-based languages. The main goal of this study is to use the lattice hierarchy introduced in Chapter 1 to match the algorithmic and language aspects of constraint programming. One crucial observation is that the three algebraic operators ($\sqcup, \Delta, \models$) are at the heart of algorithms and also of languages' abstractions. To demonstrate this point, we developed the lattice-based semantics of CCP and CLP with these operators. We believe the semantics obtained is closer to the algorithmic reality of constraint programming and thus closer to concrete implementations. This also allows us to analyse paradoxes inside some theoretical languages (such as the entailment in CCP) due to their algorithmic complexity. We hope that it gives a common ground to constraint-based languages that seem far away from each other, and forms a first basis to study a language unifying the

capabilities of the various proposals. To conclude this chapter, we discuss three research challenges in constraint-based programming languages.

L_3 and global constraints

One of the main differences between constraint-based languages and constraints libraries is the lack of support for global constraints in the former. The success of constraint programming in industrial applications is partly due to the efficiency of global constraints. However, languages generally offer limited support to program such constraints—generally through interfaces to a lower-level language (for example in GNU Prolog [DAC12]). A step into that direction is made with B-Prolog in which global constraints can be programmed through its *action rules* language [Zho06].

Considering that a CSP is a graph where variables are nodes and constraints are the edges, it seems natural that graph-based languages could be suited to program global constraints. For example, LMNtal is a graph rewriting language building on ideas of concurrent logic programming [Ued09]. Using such declarative languages could lead to the automatic extraction of entailment checking, constraint reification [BCFP13] and solution density approximation [PQZ12] based on the sole description of the global constraint propagator.

Learning the solving algorithm

The development in machine learning of the last decade virtually impacts all corners of computer science, constraint programming included. If constraint programming can be thought as computing data from relations, then machine learning is a method for computing relations from a set of data. Although this definition might seem to oppose these two fields, they are actually complementary:

- Machine learning algorithms can find and assemble an efficient solving algorithm for a specific problem.
- Constraint programming can constrain the learning process of machine learning approaches to relevant part of the state space.

SATzilla is a notable portfolio approach based on empirical model hardness, a machine learning technique to estimate the solving time of unseen instances [XHHLB08]. In this case, it is used to select the best configuration of a SAT solver in order to solve a problem. An example of the other way around is data mining with constraints where they use constraints to model mining problems [GDN$^+$15].

A challenging goal is to be able to learn over the space of search strategies, instead of using a set of existing solvers. We hope that the lattice hierarchy might be a first step towards learning how to design a constraint solver from scratch for a particular class of problems.

Beyond search in constraint programming

Searching a state space is a fundamental concept in computer science that occurs in many different settings. As a first example, constraint programming has many neighbourhood fields—that we overlooked—including SAT, (mixed)-integer programming and SMT solvers that have their own search algorithms. Also, a constraint model can be solved with other algorithms offering different characteristics, most notably local search that performs a non-exhaustive search—it does not always guarantee to find an (optimal) solution but if it does, it is usually faster than exhaustive algorithms.

Localizer, Comet and Salsa are languages supporting local search. The main difference between this paradigm and exhaustive search is that local search always manipulates grounded variables. Moving from one state to another is performed by moving to a neighbourhood state in order to reduce some violation measures. In local search, the delta operator Δ is very important to evaluate the best move among competing ones. In this respect, constraints—defined over ground values—are optimized to answer efficiently delta requests; differentiable objects are such a technique in Comet. Interestingly, the syntax of Salsa to define moves is similar for both branching and local search. Although Comet is primarily designed for local search, it also integrates search operators from OPL which allow users to mix both local and global search. All in all, although local search is a technique very different from global search, there is a substantial overlap in the combinators used in both search.

On a larger scale, search is also fundamental in the field of software verification. Model checking is a method to model a program and to verify if it satisfies some properties. Of course, exploring all the possible paths of a program generates a huge state space. Therefore, search strategies can help to increase the efficiency in this context too. Another example is the automatic theorem prover where the user is collaborating with a software to prove some theorems—instead of being fully automatic as in model checking. In this case, the state space is generated by the various ways to prove a theorem, and the user directs the search with the notion of *tactic*.

Another relevant field is rewriting systems that apply some rules reducing a term (or other entity) until some conditions are met. In the case of a non-deterministic rewriting system—two rules can be applied at the same time—the search strategy can be crucial to ensure the termination of the system. For example, in ELAN, they propose a search strategy language [CB00] based on a combinator approach that is very similar to the ones we overview in Section 2.6.3.

The fields of constraint programming, verification and rewriting systems are only a few of the numerous examples where search is actively used. A key observation to establish the "perfect set of search operators" is to realize that these operators are similar across fields although the underlying state space is not. Therefore, we believe that operators over the lattices L_4 and above lay the foundations for such a language.

Part II

Spacetime Programming

Chapter 3

Overview of Spacetime Programming

This chapter introduces the motivations, syntax and intuitive semantics behind a new language called spacetime programming or "spacetime" for short. Spacetime is based on synchronous programming [Hal92] and concurrent constraint programming (CCP) [SR89]. We combine these two paradigms to tackle the compositionality issues of search strategies discussed in Section 2.6. Firstly, we rely on the notion of logical time of the synchronous paradigm to coordinate and compose search strategies. Secondly, we build on the CCP's model of partial information to define the variables of a program as lattice structures. It is instrumental to enable search strategies to communicate during the exploration of the search tree.

We first introduce the synchronous model of computation and the language Esterel in Section 3.1. We then merge the *linear* model of time of Esterel with *branching* time in order to program backtracking algorithms. Blending linear and branching time shapes the model of computation of spacetime programming (Section 3.2). We introduce its syntax and intuitive semantics in Section 3.3. In Section 3.4, we give two examples of search strategies in spacetime that are specialized to the layer L_4 in the lattice hierarchy. Thereafter, we extend spacetime with the concept of hierarchical computation to program layers above L_4 (Section 3.5). It is exemplified on two search strategies in L_5 in Section 3.6.

In sum, we contribute to extending the synchronous language Esterel with lattice-based variables, branching time and hierarchical computations.

3.1 Synchronous programming

The concept of time is recognized as highly difficult to define. The Newtonian view is the most intuitive: it defines time as a quantity that can be divided into very small and indivisible units. According to this view, these "smallest units of time" define a global clock where every tick makes all the atoms of the universe progress at once. A second observation is that time is related to space by the notion of causality. For example, a plane can only be in the sky (event B) if it has first taken off (event A). It is assumed that whenever the existence of an event B depends on

the one of A, then A must occur before B. A global clock is convenient to define this "happens-before" relationship and the notion of simultaneous events: it is directly mapped onto the order given by the global clock. Synchronous programming brings the notion of linear time, where everything happens before or after something else, and causality inside a programming paradigm.

3.1.1 Linear time

The synchronous paradigm [Hal92] was initially designed for modeling systems reacting to simultaneous events of the environment—different inputs can arrive at the same time—while avoiding typical issues of parallelism, such as deadlock or indeterminism. A simple example is a watch: its state changes when the user presses buttons or when it receives a signal indicating that a second has elapsed. The main idea of this paradigm is to propose a notion of logical time dividing the execution of a program into a sequence of discrete instants that are conceptually instantaneous. This assumption of instantaneous computation is called the *synchrony hypothesis*.

We explain the synchronous paradigm with the language Esterel [Ber00b] because spacetime is based on its syntax and model of computation. An Esterel program reacts to input signals and produces outputs. A signal is a variable with a boolean type indicating if it is present or absent. We illustrate the synchronous behavior of a program with a variant of the standard ABRO example [Ber00a].

Example 3.1 (ABO)**.** When the signals A and B are received, the output O is emitted. The signal O carries an integer value indicating the number of times it has been emitted. The following program is given in Esterel V5 [Ber00a]:

```
module ABO:
  input A, B;
  output O := 0: integer;
  loop
    [ await A || await B ];
    emit O(pre(?O) + 1);
    pause;
  end loop
end module
```

We explain the program from its innermost instructions:

- The statement `await A` indefinitely waits for the signal A and terminates when A occurs.
- The parallel composition P `||` Q concurrently and cooperatively executes two processes and terminates when both are terminated. In the example, it terminates as soon as both signals A and B have occurred.
- Once terminated, the signal O is emitted and can be retrieved by the user to activate, for example, a real world command.

- We initialize the value of O to 0 and increment it when emitted, and then we use `pre(?O)` to retrieve the value of O in the previous instant.
- The instruction `loop` p `end loop` executes indefinitely the process p. We forbid the body of the loop to be instantaneous due to the synchrony hypothesis.
- Hence, to avoid infinite loop within one instant, we must delay each iteration to the next instant with the statement `pause`.
- The whole behavior is reset at each loop iteration.

⌟

Operationally, we can view a synchronous program as a coroutine: a function that can be called multiple times and that maintains its state between calls. When the program is called, its code is resumed from the last `pause` statements reached. This temporal dimension opens the door to two different kind of memories:

- The persistent memory for the values spanning several instants, for example the integer value carried by the signal O.
- The local memory for those only relevant to a single instant, for example the signals A and B.

User inputs can be injected into the program in-between two resumptions of the program. The external inputs are collected within a host language from which one instant of the synchronous program is called.

The well-defined semantics of synchronous languages for the treatments of simultaneous events has a wide variety of applications encompassing interactive mixed music [BJMP13], dynamic network protocol [MB05] and web programming [BS14, ESC16], just to name a few of them.

3.1.2 Causality analysis

A programming language is a set of concepts defining how the space of a program evolves through time under some laws. Concretely, the space is the memory storage, the laws are the code of the program and time is the causal application of the laws to the space. The space and laws components in languages are well-studied but time is often left implicit. An explanation is that most programs are written in a sequential fashion, and thus they are causal by default since they do not describe simultaneous events. Therefore, time and simultaneous events are generally not supported explicitly. The most widespread attempt to support these notions in sequential languages is the mechanism of threads. However, it is widely acknowledged that threads are difficult to program correctly and to debug [Lee06]. From our point of view, the problem with threads is that the causality principle must be implemented manually by the programmer, which is usually not an easy task.

The synchrony hypothesis enables compile-time static analysis to verify the cooperative and correct behaviour of processes. Cooperative execution means that the processes do not compete for a resource—unlike in many multi-threaded architectures where race conditions can occur. At this stage, it is important to distinguish between *parallel* and *concurrent* programs. The parallel statement in Esterel is compiled into sequential code in which the processes are statically interleaved. Therefore, our model of computation is concurrent (despite the name of the operator `||`) but the execution of the program is sequential[1].

We argue that time is a central notion to the concurrent execution of processes. It is implemented in synchronous languages using a global clock for coordinating the evolution of the processes. During an instant, every process instantaneously makes a step forward regardless of the other processes. Therefore, we must verify that every interleaving of two processes P and Q always leads to the exact same output, and thus that the computation is deterministic. This is the role of the causality analysis. However, this comes with an important cost: the causal relation of the program during any instant must be decidable, this implies that an instant is bounded in space and time. To illustrate causality, we consider a non-causal Esterel program:

```
if S then emit R else emit S end
```

where `if` tests the presence of a signal. We read this program as: "if S is absent emit S otherwise emit R". The causality issue comes from the fact that if S is absent then it is present. Of course, it is never present and absent at the same time according to the structure of the program, but under the synchrony hypothesis it has these two possible statuses. Therefore, the causality analysis will reject this program.

Finally, we consider a pseudo-program where boolean signals are replaced by variables over the lattice of increasing integer $LMax$.

```
if x ⊨ y then
  y = y ⊔ (y + 1)
end
```

We read this program as: "if we can deduce y from x, then increase y by one". This simple process shows a non-causal relationship inside one instant. To show it, we admit that x and y are both equal to 1 at the beginning of the instant. When the condition $x \models y$ is checked, x and y equal 1, therefore the condition is $true$. However, when executing the body, we increment the value of y which makes the condition $false$ since $1 \not\models 2$. Since we made the hypothesis that the computation is instantaneous, this program is not causal and we must reject it.

The language of the latest example is very similar to CCP with indexicals (Section 2.4.2). We claim that the causality analysis is a bridge between CCP and synchronous programming. Specifically, the causality analysis of Esterel is similar

[1] The kind of parallelism in the synchronous paradigm is similar to *and-parallelism* in logic languages, therefore it is a challenging problem to obtain true parallelism.

to the monotonicity analysis of CCP with indexicals. Based on this observation, we develop a causality analysis for lattices-based programs in Section 4.5.

3.2 Concepts of spacetime programming

This section is dedicated to explaining the model of computation of spacetime programming. We discussed about the synchronous language Esterel which is constituted of boolean variables[2] and time progressing linearly. In spacetime, we depart from Esterel by proposing the notion of branching time to program backtracking algorithms (Section 3.2.1) and lattice-based variables (Section 3.2.3). An innovation in spacetime is to blend linear and branching time together in a single language (Section 3.2.2).

3.2.1 Branching time

Until now, we assumed that time is totally ordered: a point in time t always happens before or after a point in time t'. Alternatively defined, the space of a program has a single past and a single future. We challenge this notion by investigating time as a partial order. At a point in time t, the program has the ability to create two distinct and unordered futures t' and t'' such that $t < t'$ and $t < t''$. If the space is duplicated at some point in time, then the two new spaces become causally independent and unordered through time. We refer to this concept as branching time, and we call the succession of events leading to a particular space a *world line*.

Branching time is pervasive in computer science with backtracking algorithms: several choices are successively made until we find a solution. This succession of choices is a world line. A world line is therefore the evolution of the variables through time, and backtracking is the mechanism for moving between world lines. We implement backtracking with a queueing strategy (Definition 1.5.2) that stores the world lines. We conciliate the exploration of the search tree and the time dimension with the following principle:

A node of the search tree is explored in exactly one logical instant.

Therefore, the extraction function *pop* of the queuing strategy is called at the beginning of an instant and the function *push* is called at the end of an instant. In this respect, a second principle is:

A search strategy is a synchronous process.

Therefore, search strategies are synchronized through time and progress at the same rate when exploring the search tree.

[2] Although values can be carried by signals in Esterel, they have a limited integration into its concurrency model (see Section 4.6).

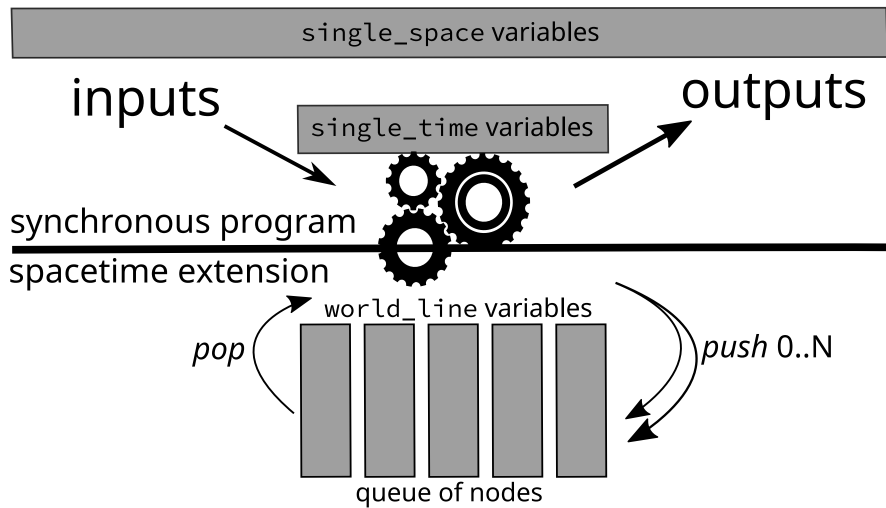


Figure 3.1: Spacetime extension of the synchronous model of computation.

3.2.2 Blending linear and branching time

The innovation in spacetime is to propose linear and branching time blended in the same language. We notice that time is passing because we can observe the space of the program changing. Instead of trying to propose abstractions to modify the flow of time, we choose to work on the space of the program directly. Therefore, we propose to annotate the variables with a *spacetime attribute* to indicate how a variable evolves in space and time. For this purpose, a spacetime program has three distinct memories in which the variables can be stored:

(1) Local memory (keyword `single_time`) for variables local to an instant and re-allocated in each node. A `single_time` variable only exists in one instant.

(2) Global memory (keyword `single_space`) for variables evolving globally to the search tree. A `single_space` variable has a unique location in memory throughout the execution. For example, we store the number of nodes of the search tree in a `single_space` variable.

(3) Backtrackable memory (keyword `world_line`) for variables local to a path in the search tree.

The latter memory stores copies[3] of the variables when a choice has been encountered. More precisely, the backtrackable memory is part of the queue of nodes representing the remaining part of the search tree to explore.

[3]Besides copying, different variable restoration policies can be used such as trailing or recomputation. Nevertheless, from the spacetime point of view, the restoration policy is abstracted in the implementation of the corresponding lattice.

We depict the model of computation of spacetime in Figure 3.1. We extend the synchronous model of computation with a queue of nodes and an abstract queueing strategy. We will see in Section 3.5 that we can explicitly parametrize the computation with a user-defined queueing strategy. The queue is manipulated with the functions *pop* and *push*. At the beginning of each instant we instantiate the `world_line` variables with the values stored in the node popped from the queue. At the end of each instant, we push an arbitrary but bounded number of nodes onto the queue. For examples, pushing two nodes in each instant creates a binary search tree, and pushing an empty set of nodes has the effect of pruning the subtree of the current node.

3.2.3 Host variables as lattice structures

We propose to extend the domains of the variables in the synchronous paradigm to lattice structures. It is inspired by concurrent constraint programming (CCP) [SR89] (see Section 2.4.2). In CCP, the processes interact through a shared store of constraints with two instructions metaphorically called `ask` c and `tell` c. The instruction `ask` c succeeds if we can deduce the constraint c from the global store (entailment operation), the instruction `tell` c adds the constraint c onto the global store (join operation). In contrast, spacetime embeds this communication mechanism inside each variable instead of considering a global and shared constraint store. The key idea is to suppose that every variable has a lattice structure and implements the entailment and join operations.

However, spacetime is not a general purpose language and it relies on a host language for the definition of the lattices. Similarly to many synchronous languages, spacetime needs to be embedded into a host language. Firstly, it simplifies the design of the core language since many required programming concepts ranging from arithmetic expressions to data aggregation in modules are already provided in the host language. Secondly, it facilitates the integration of a spacetime program in a larger project. The synchronous program can react on inputs from the environment such as graphical user interface and the network, and the user can activate the next instant on-demand. These are important ingredients for programming interactive systems with the spacetime paradigm.

We embed spacetime inside the object-oriented programming language Java. It allows users to declare processes and spacetime variables in a Java class along to Java methods and attributes. This is inspired by Bloom^L [CMA$^+$12], a language for distributed computing using lattices. Similarly to Bloom^L, we leave the definitions of the lattices to the host language which must only expose monotonic functions over these lattices. This enables us to use Java classes and libraries directly in spacetime.

We implemented a prototype compiler of the spacetime language[4] in Rust. We compile a spacetime program to an extension of the synchronous library Sugar-

[4]The compiler is publicly available at `https://github.com/ptal/bonsai`.

Cubes [BS00] in Java. We provide a library of lattice structures such as $LMax$ and $LMin$, and the user can define its own lattice structures as long as it implements the right interfaces. In particular, we encapsulate the constraint solver Choco [PFL15] in a lattice-based variable implementing the lattice L_3. This abstraction over Choco is only 300 lines of code which hints that it would not be prohibitive to support other Java constraint solvers.

3.3 Syntax and intuitive semantics

We organise the statements of spacetime into fragments, each relevant to a part of the lattice hierarchy and to the compositional semantics of search strategies. As a reference sheet, we give the full syntax of spacetime in Figures 3.2 and 3.3, and we explain each instruction.

3.3.1 Communication fragment

The communication fragment of spacetime allows two processes to communicate over a shared variable. In order to keep the communications monotonic, a process must sometimes annotate how the variable is accessed (in read, write or read-write modes). Access modes are required when calling external functions from the host language. It acts as the promise that the host code will manipulate the variables following the contract. Unfortunately, the verification that the contract is fully respected is left to the user and the host language. We define the spacetime expressions as follows:

- `bot` and `top` respectively represent the bottom and top element of their corresponding lattice.
- `read` x, `write` x and `readwrite` x specify an access mode to the variable x. The access mode indicates how the variable is manipulated in the host functions. This is especially useful for the compiler to verify that a program is causal (see Section 4.5).
- `pre` x gives a stream interpretation to the variable x and returns the value of the variable at *its* previous instant. At the first instant of a variable x, we have `pre` x `= bot`. The `pre` operator is similar to the one used in the dataflow synchronous paradigm [BCLGH93]. However, it behaves differently depending on the spacetime of x:
  - `single_time`: it cannot be applied to single time variables since, conceptually, these variables only exist in a single instant.
  - `single_space`: it works similarly to dataflow language, the value of `pre` x is the one of x in the previous instant.

```
⟨proc⟩       ::= proc ⟨ident⟩ ⟨params⟩? = ⟨p⟩                         (process declaration)

⟨p, q, ...⟩ ::=                              communication fragment (L3 and below)
             | ⟨var_decl⟩                                          (variable declaration)
             | when ⟨trilean⟩ then p else q end                                     (ask)
             | ⟨var⟩ <- ⟨hexpr⟩                                                     (tell)
             | ⟨call⟩                                                      (function call)
             |                                                      synchronous fragment
             | nothing                                                   (empty process)
             | pause                                                             (delay)
             | par p || q end                          (disjunctive parallel composition)
             | par p <> q end                          (conjunctive parallel composition)
             | p ; q                                            (sequential composition)
             | loop p end                                                (infinite loop)
             | abort when ⟨trilean⟩ in p end                                  (abortion)
             | suspend when ⟨trilean⟩ in p end                              (suspension)
             | for ( ⟨combinator⟩ ) ( ⟨range⟩ ) p end                (instantaneous loop)
             |                                                search tree fragment (L4)
             | space p end                                               (branch creation)
             | prune                                                      (branch pruning)
             |                                         universe fragment (L5 and above)
             | universe ⟨uparams⟩ (with ⟨ident⟩)? in p end             (universe creation)
             | pause up                                       (delay in the upper universe)
             | stop                                       (delay in the outermost universe)

⟨spacetime⟩ ::= single_space | world_line | single_time

⟨combinator⟩ ::= || | <> | ;

⟨var_decl⟩ ::= ⟨spacetime⟩ ⟨type⟩ ⟨ident⟩

⟨range⟩   ::= ⟨var_decl⟩ : ⟨hexpr⟩

⟨call⟩    ::= ⟨hexpr⟩ . ⟨ident⟩ ⟨args⟩                                    (method call)
           |  ⟨ident⟩ ⟨args⟩                                            (function call)

⟨args⟩    ::= ( ⟨hexpr⟩ % , )

⟨params⟩ ::= ( ⟨var_decl⟩ % , )

⟨uparams⟩ ::= ( (⟨spacetime⟩ ⟨ident⟩) % , )
```

Figure 3.2: Syntax of the spacetime language. We are using the non-standard notation $e?$ for optional expression and $e\%e'$ for a possibly empty list of e separated by e'. The rules ⟨*var*⟩, ⟨*ident*⟩ and ⟨*type*⟩ are host defined expressions.

⟨*hexpr*⟩	::=	⟨*expr*⟩	
	|	⟨*call*⟩ | ⟨*host_expr*⟩	(host interface)
	|	⟨*access*⟩ ⟨*var*⟩	(restricted variable access)
⟨*expr*⟩	::=	**bot** | **top** | ⟨*trilean*⟩	(lattice expressions)
	|	**pre*** ⟨*var*⟩	(stream variable)
	|	⟨*access*⟩ ⟨*var*⟩ `::` ⟨*var*⟩	(bottom up transfer)
	|	`(` ⟨*expr*⟩ `)`	
⟨*trilean*⟩	::=	**true** | **false** | **unknown**	
	|	⟨*expr*⟩ `|=` ⟨*expr*⟩	(entailment)
	|	⟨*trilean*⟩ **and** ⟨*trilean*⟩	(conjunction)
	|	⟨*trilean*⟩ **or** ⟨*trilean*⟩	(disjunction)
	|	**not** ⟨*trilean*⟩	(negation)
⟨*access*⟩	::=	**read** | **write** | **readwrite**	

Figure 3.3: Syntax of the expression of the spacetime language. We use $e*$ for repeating e zero or more times. The rule ⟨*host_expr*⟩ is a host defined expression.

  – `world_line`: it returns the value of the variable x obtained in the parent of the current node.

The expressions are used with the following statements:

- *spacetime Type var* declares a variable *var* of type *Type*. The attribute *spacetime* situates the variables in a specific memory as explained in Section 3.2.2. When executed, the variable is initialized to the bottom value $\bot$ of its lattice *Type*.
- `when` a `|=` b `then` p `else` q `end` executes the body p whenever we can deduce b from a in the current instant (entailment relation). The process p is executed when the condition maps to *true* and otherwise q is executed. It is possible to use any three-valued logic expression instead of a `|=` b (see below for the truth table).
- x `<-` e augments the information in x with the expression e; it performs the join operation $x = x \sqcup e$.
- $o.m(a_1, \ldots, a_n)$ calls the host method m on the object o with the arguments $a_1, \ldots, a_n$. We also support the function call syntax without a target object.

A	B	`not` A	A `and` B	A `or` B
true	true	false	true	true
false	true	true	false	true
false	false	true	false	false
unknown	true	unknown	unknown	true
unknown	false	unknown	false	unknown
unknown	unknown	unknown	unknown	unknown

Table 3.1: Truth table of the Kleene three-valued logic used in spacetime.

Closed-world assumption

A problem when querying a system with a negative request is that even if a piece of information is not yet present, it can become present in the future. A classic workaround is the closed-world assumption: we make the assumption that, at the time of the request, we know everything. Notably, this semantics and others have been extensively studied in Prolog under the negation as failure principle. The synchronous model of time provides a semantics of negative knowledge. Thanks to the synchrony hypothesis, we know everything about an instant, and complications to deal with negative knowledge do not arise. However, we still need to take into account that some pieces of information are unknown during the current instant as explained in the next section.

3.3.2 Undefinedness in three-valued logic

Spacetime is built on the Kleene three-valued logic which is similar to boolean logic with the additional undefined value—represented by `unknown` in spacetime. The truth table of the three logic operators provided in spacetime is given in Table 3.1. A problem in languages with a three-valued logic is to deal with the unknown value. Consider the following program:

```
when x |= y then P else Q end
```

The case where $x \models y$ maps to *unknown* is often problematic: should we execute the else-branch or nothing at all? This problem stems from the conflicting semantics between expressions using a three-valued logic and the `when` statement defined over a boolean logic. A solution is to promote *unknown* to $false$ in the condition of `when` and, in this example, to execute the process Q. We call this mapping the closed-world assumption (CWA) as it relates to the CWA in logic programming (see 3.3.1 above).

A second problem, highlighted in the relational semantics of constraint modelling languages [FS09], occurs with nested expressions. How should the unknown value be propagated inside an expression? We illustrate it with the following program:

```
when not (x |= y) then P else Q end
```

Should we activate the CWA at the level of the expression $x \models y$ or at the level of the condition $\neg(x \models y)$? Assuming that $x \models y$ maps to *unknown*, it leads to two possible semantics:

1. The CWA is activated as soon as possible in which case $x \models y$ maps to $false$ and $\neg false = true$, thus P is executed.

2. The CWA is activated as late as possible in which case $\neg(x \models y)$ maps to *unknown* which is promoted to $false$, thus Q is executed.

In [FS09], these two options are respectively called the relational and Kleene semantics. In the relational semantics, the logic operators are boolean since the unknown value is always transformed at the value-level. In contrast, the Kleene semantics makes use of the Kleene logic operators to compute the result (as introduced in Table 3.1). Our solution is to use the entailment as a logic operator to explicitly indicate when to perform CWA. Given the entailment lattice ES (Definition 1.40), the truth table of $\models_{ES}$ is generated as follows:

$$a \models_{ES} b \mapsto \begin{cases} true & \text{if } a = b \vee b = unknown \\ false & \text{otherwise} \end{cases}$$

A key observation is that $a \models_{ES} b$ cannot return *unknown*. Therefore, the CWA over an expression $x \models y$ can be manually triggered using $(x \models y) \models_{ES} true$—if $x \models y$ equals *unknown*, it will be promoted to $false$ since $unknown \models_{ES} true \mapsto false$. The relational and Kleene semantics can be obtained with the following programs:

```
when not ((x |= y) |= true) then P else Q end        (Relational)
when (not (x |= y)) |= true then P else Q end        (Kleene)
```

These two semantics are compatible in our language due to the logic interpretation of the entailment operator. Also, this is due to the backward compatibility of the Kleene logic operators with the boolean truth tables when we remove the value *unknown*. In order to simplify the programs, the conditional statements implement a Kleene semantics by default, and therefore the second condition can be written more simply as `not (x |= y)`.

3.3.3 Synchronous fragment

The temporal dimension in synchronous languages opens the door to many primitives for controlling the state of a program. For example, when considering time, the traditional conditional statement "if-then-else" has several variants including `when`, `abort` and `suspend`. Spacetime is based on Esterel [Ber00a] for most of these

statements and on `Quartz` [Sch09] for the conjunctive parallel `<>` and the instantaneous loop `for`. The statements in the synchronous fragment of spacetime are interpreted as follows:

- `nothing` describes the empty process[5] that does nothing and terminates instantaneously.
- `pause` delays the execution of the process to the next instant.
- `par` p `||` q `end` is the parallel composition of the two statements p and q—called processes in this context. It terminates when both p *and* q are terminated.
- `par` p `<>` q `end` is similar to `||` with the difference that it terminates whenever p *or* q terminates.
- p `;` q is the sequential composition of the two statements p and q. It executes p and when it terminates, it executes q.
- `loop` p `end` executes indefinitely the body p. An instant must always be executed in bounded time, and so to prevent infinite loop within an instant, the body p must contain a `pause` statement[6].
- `abort when` *trilean* `in` p `end` executes p until the condition *trilean* is equal to *true* (in the lattice ES). It can be imagined as a version of `when` where the condition is rechecked in every instant but the program state of p is retained. It does not execute its body p in the instant where *trilean* is equal to $false$.
- `suspend when` *trilean* `in` p `end` suspends the execution of p whenever the condition *trilean* is mapped to *true*. It only suspends its execution during the current instant and resumes p as soon as the condition *trilean* is not equal to *true* anymore.
- `for(;)(<var_decl> : r)` p `end` generates the sequence p_1 `;` p_2 `;` $\dots$ `;` p_n where n is the size of the range r. Additionally, we can generate a conjunctive and disjunctive parallel sequence using `for(||)` and `for(<>)`. In each iteration, a value is extracted from r and stored in the variable occurring in `<var_decl>`. We restrict the body p to be instantaneous, and we leave temporal processes involving iteration to the statement `loop`.

3.3.4 Search tree fragment

An important contribution in spacetime is the operators for dynamically creating and pruning a search tree. More precisely, in each instant we have a store of

[5] Also known as `skip` or $\emptyset$ in some process algebras.

[6] This can be statically checked with a compile-time analysis [TDS03].

children nodes—similar to the lattice $Store(L_3)$—which is manipulated with `space` and `prune`. At the end of an instant, this store is pushed onto the queue, and the children nodes will be executed in turn in the future instants. This extension to synchronous languages is provided with the following statements:

- `space` p `end` creates a branch from the current node to a child node. The process p is executed when the child node is instantiated in a future instant.
- `prune` indicates that a part of the subtree of the current node should not be explored. This statement alone creates a *pruned branch* which might have the effect to discard one or more `space` statements encountered in the current instant.

The `space` and `prune` statements come with a compositional semantics regarding the sequence and the parallel operators which is formally introduced in Section 4.2. For now, it is sufficient to be aware that:

(i) `space` creates a branch and `prune` cancels a branch,

(ii) two branches in a sequence are concatenated,

(iii) two branches b_1 and b_2 created in parallel processes are merged together with b_1 `||` b_2 and b_1 `<>` b_2, and

(iv) `prune ||` p is equivalent to p, and `prune <>` p is equivalent to `prune`.

3.3.5 Read-write scheduling

To ensure determinism of the computation, we must read every variable when they have been totally written unless it does not impact the monotonicity of the program. We illustrate this with an example.

Example 3.2 (Process scheduling)**.** We initialize two counters: x to 1 and y to 0. Three processes are launched in parallel: P joins x with 2 or 3 if we can deduce or not x from y, Q increases the value of y to 2, and R prints the values of x and y.

```
proc scheduling =
  single_time LMax x = new LMax(1);
  single_time LMax y = new LMax(0);
  par
  || when x |= y then x <- 2 else y <- 3 end                (P)
  || y <- 2                                                 (Q)
  || System.out.println("x=" + read x + " y=" + read y)     (R)
  end
```

The program deterministically produces a unique answer which is `"x=1 y=3"`. Firstly, R cannot be executed until all writes into x and y have happened, so R is executed last. Next, P might be scheduled first but it is then blocked in

the condition because at this point we still must write into y. Therefore, the process Q is scheduled: it joins y with 2 and terminates. Intuitively, we can execute the entailment condition in P once we are sure that further execution of the program will not change the entailment result in the current instant. Now, we have $1 \models 2 \mapsto \mathit{false}$, and thus we can execute the else-branch because no more write can be triggered in x and change this result. Finally, the values can be printed since all possible writes into x and y have been executed. ⌟

Design rational 1: Pure entailment condition

We do not permit host functions inside the entailment condition neither do we permit write and read-write accesses. This is because the entailment can be checked several times during one instant but a host function should only be called once. First, it is not guaranteed that the host function is idempotent ($f(f(x)) = f(x)$), and thus that determinism would be preserved. And second, host functions can generate side effects and it is usually difficult to use them if they can be called several times.

3.3.6 Derived statements

We presented the kernel of spacetime. For convenience, we extend it with other statement that are derived from the kernel statements. We first derive several expressions using the three-valued logic and the entailment operator:

$$
\begin{array}{lcll}
e \texttt{ == } e' & \stackrel{\text{def}}{=} & e \texttt{ |= } e' \texttt{ and } e' \texttt{ |= } e & \text{(equality)} \\
e \texttt{ != } e' & \stackrel{\text{def}}{=} & \texttt{not (}e \texttt{ == } e'\texttt{)} & \text{(disequality)} \\
e \texttt{ |< } e' & \stackrel{\text{def}}{=} & e \texttt{ |= } e' \texttt{ and } e \texttt{ != } e' & \text{(strict entailment)}
\end{array}
$$

Note that these operators are constructive: an unknown value on the left or right of the derived `and` or `not` expression is propagated up.

We next introduce the flow statement and flow process that encapsulate `loop` and `pause` into a single abstraction.

$$
\begin{array}{lcll}
\texttt{flow } p \texttt{ end} & \stackrel{\text{def}}{=} & \texttt{loop } p \texttt{ ; pause end} & \text{(flow)} \\
\texttt{flow } f \texttt{ = } p & \stackrel{\text{def}}{=} & \texttt{proc } f \texttt{ = flow } p \texttt{ end} & \text{(flow process)}
\end{array}
$$

The pre operator can be applied to an expression which turns all the variables of the expressions into their "pre" versions:

$$
\texttt{pre(}e\texttt{)} \stackrel{\text{def}}{=} \text{e}[\mathit{var} \mapsto \texttt{pre } \mathit{var}] \quad \text{(pre expression)}
$$

The substitution $e[\mathit{var} \mapsto \texttt{pre } \mathit{var}]$ is defined inductively over the expressions.

The conditional statement with a single branch is useful as well:

`when` e `then` p `end` $\stackrel{\text{def}}{=}$ `when` e `then` p `else nothing end` (unary when)

Esterel proposes a weak version of the abortion for delaying the abortion by one instant. The statement `weak abort when` e `in` p `end` is similar to `abort` but when e maps to *true* we execute the program p "one last time". It can be derived from `abort` by using `pre` over the condition e.

`weak abort when` e `in` p `end` $\stackrel{\text{def}}{=}$ `abort when pre(`e`) in` p `end` (weak abort)

Finally, in the kernel language, the variables cannot be initialized when declared. This is partly because the initialization scheme differs depending on the spacetime attribute.

(`single_space` | `world_line`) $Type\ x = e$ $\stackrel{\text{def}}{=}$ (variable initialization)
    (`single_space` | `world_line`) $Type\ x; x$ `<-` e

`single_time` $Type$ x `=` e `;` p $\stackrel{\text{def}}{=}$ (persistent initialization)
    `single_time` $Type$ x`;`
    `par` p `<> flow` x `<-` e `end`

For the single space and world line variables, the initialization is working as expected because they are not reinitialized during their lifetimes. However single time variable are reallocated at a fresh location and reinitialized to $\bot$ between each instant. When initializing such variables to an expression, the most rational design is to reinitialize these variables at the beginning of each instant with this expression instead of $\bot$. For this purpose, we create a flow reinitializing the variable in every instant. It is composed with a conjunctive parallel: as soon as p is terminated, the variable must exit its scope and its initialization flow terminates accordingly.

3.4 Programming search strategies in L_4

We present a first and complete spacetime program of the "propagate-and-search" algorithm as well as the branch and bound algorithm for constraint optimization problems.

3.4.1 A minimal constraint search algorithm

In Section 1.5.4, we saw that an exploration strategy is the composition of a inference and branching strategies. We program a minimal search component of a constraint solver that is extended all along this chapter and Chapter 6. We first

introduce the propagate and search algorithm, and then define the branching strategy. This first spacetime program implements the exploration strategy generating the "raw search tree". Once this search tree is defined, we can prune and modify its exploration as shown in the next sections.

Propagate and search algorithm

Following the definition of a CSP as a couple $\langle d, P \rangle$ in the lattice L_3 (Section 1.3.4), we define the class `Solver` with two spacetime attributes `domains` and `constraints`. We compose the inference engine and the branching strategy with a parallel operator, and we abort the computation whenever we reach a solution.

```
class Solver {
  world_line VStore domains;
  world_line CStore constraints;
  single_time ES consistent;

  public Solver(VStore domains,
   CStore constraints) {
    this.domains = domains;
    this.constraints = constraints;
  }

  flow propagation =
    consistency <- read constraints.propagate(readwrite domains);
    when unknown |< consistency then
      prune;
    end

  proc fail_first_middle =
    single_space FailFirstMiddle brancher =
      new FailFirstMiddle(domains, constraints, consistent);
    brancher.branch()

  public proc search =
    par propagation() <> fail_first_middle() end

  public proc first_solution =
    weak abort when consistent in
      search()
    end
}
```

The solver's variables have a `world_line` spacetime because they are attached to a path of the search tree and need to be restored upon backtracking. Also, these two fields bridge to the lattice L_3 that is implemented in the host language. Their types, respectively `VStore` and `CStore`, are Java classes interfacing with the `Choco` constraint solver [PFL15]. In addition, we use a variable `consistent` reflecting the consistency of the current CSP. The class `Solver` exhibits four main components:

- `propagation` narrows the domains in each node. Therefore, we read and write on the domains of the variables but only read the constraint store. It uses the propagation engine of `Choco` implemented in the method `propagate` of the constraint store. Whenever the CSP is either failed or has a solution, we prune the current subtree because we do not want to explore the subtree of this CSP.

- `fail_first_middle` models the branching strategy (see below).

- `search` combines the former two: it models a strategy exploring the full search tree.

- `first_solution` uses `weak abort` for terminating the search when the first solution is found. It terminates the program immediately on the next instant of a solution node.

Depending on the user needs, either the process `search` or `first_solution` can be called on this class. We explain the `fail_first_middle` branching strategy in the following section.

Branching strategy

In Section 1.5.3, we divided the branching into three functions for splitting the state space of a constraint problem. The class `FailFirstMiddle` implements a branching strategy that splits in the middle a variable x with a value v ($x \leq v \vee x > v$) such that x is the first non-assigned variable with the smallest domain.

```
class FailFirstMiddle {
  world_line VStore domains;
  world_line CStore constraints;

  public FailFirstMiddle(VStore domains,
   CStore constraints, ES consistent) {
    this.domains = domains;
    this.constraints = constraints;
  }

  public flow branch =
    single_time IntVar x = failFirstVar(read domains);
    single_time Integer v = middleValue(x);
    space constraints <- x.le(v) end;
    space constraints <- x.gt(v) end

  // Interface to the Choco solver.
  private failFirstVar(VStore domains) { ... }
  private middleValue(IntVar domains) { ... }
}
```

Similarly to `Solver`, we initialize the class with the corresponding CSP. We build the search tree using two Java function for selecting the variable and its value to split on. The search tree is then built with two `space` statements where the first describes a future where $x \leq v$ and the second a future where $x > v$. We use the interface of the solver `Choco` to create these constraints.

The process `branch` repeatedly splits the search space in every node. As an example of processes' communication in one instant, we read `domains` after the propagation happened. Therefore, the two processes (indirectly) communicate over the variable `domains`. In general terms, we wait for all write operations to happen on the variables—here `domains` and `constraints`—before reading into it.

3.4.2 Branch and bound

As we have seen in Section 1.6, the branch and bound algorithm constrains the search tree (lattice L_4) instead of the CSP (lattice L_3). In this section, we show that we can program this algorithm in spacetime. We propose the class `MinimizeBAB` for minimizing a variable x:

```
class MinimizeBAB {
  world_line VStore domains;
  world_line CStore constraints;
  single_space IntVar x;
  single_space CStore obj = bot;
  single_space LMax obj_ver = bot;
  world_line LMax con_ver = new LMax(0);

  public MinimizeBAB(VStore domains, CStore constraints, IntVar x) { ... }

  public proc search =
    par
    || minimize()
    || yield_objective()
    end

  flow minimize =
    single_time ES consistent = read constraints.consistent(read domains);
    when consistent == true then
      obj <- read x.lt(read x.getValue());
      readwrite obj_ver.inc();
    end

  flow yield_objective =
    when pre (con_ver |< obj_ver) then
      constraints <- obj;
      con_ver <- obj_ver;
    end
}
```

Along with the current CSP, we define the constraint store `obj` to store the objective constraints and the variable `x` that is to be optimized. The `single_space` specifier indicates that the constraint store `obj` is global to the search tree, and thus will not be backtracked. The class has two main processes:

(i) `minimize` which, in each solution node, strengthens the current objective value with the constraint $x < v$ where v is the current value of x, and

(ii) `yield_objective` that ensures the new bounds of the objective function are taken into account in the main store `constraints`.

In theory, since `constraints` is a lattice, we have the equivalence $c \sqcup d \equiv c \sqcup d \sqcup d$. Hence, it would be correct to add the objective function inside `constraints` in each instant. However, in practice, adding duplicate constraints into the store consumes memory and their removal is time consuming. To avoid this problem, we use a versioning system to add the objective in `constraints` only when it is not already added. For this purpose we use two variables `obj_ver` and `con_ver` respectively for the versions of the objective and constraints stores. We increment the variable `obj_ver` each time we modify the objective store. Similarly, the variable `con_ver` indicates the version of the objective store currently active in `constraints`. Whenever the constraint store has a different version than the objective store, we update it with the last objective constraint.

There is an important detail to notice: when testing the version numbers, we use a `pre` expression. Interestingly, if we do not, the causality analysis will fail since we have a cyclic dependency in the data: `obj_ver` depends on `constraints` and vice versa. Fortunately, the causality analysis prevents us from having a bug: adding the current objective in a solution node would make the CSP unsatisfiable.

3.5 Hierarchy in spacetime with universes

In the former sections, we demonstrated that spacetime is effective to program the layer L_4 of our lattice hierarchy. We extend the model of computation of spacetime with *universes* to program layers above L_4, and we show how we can share and communicate information across layers.

Our universe extension synthesizes time hierarchy from synchronous programming and spatial hierarchy from logic programming. Time hierarchies were first developed in Quartz [GBS13] and ReactiveML [Pas13, MPP15] to execute a process on a faster time scale. They propose that in one instant of the global clock, we can execute more than one local step of a process. Spatial hierarchies were introduced with logic programming and more particularly with deep guards and computation spaces in the Oz programming language [Sch02]. It executes a process locally with its own queue of nodes (see Section 2.5).

We propose the statement `universe` p `end` which encapsulates the spatial and time dimensions of a computation. In the lattice hierarchy, universes enable users

to specify strategies over the lattices L_5 and above. This extension is particularly useful to program search strategies that restart the search or explore several times a search tree. We present the syntax and intuitive semantics of universes in Section 3.5.1. We then explain how two layers in the hierarchy can communicate through universes (Section 3.5.2).

3.5.1 Universe fragment

We introduce several statements for creating a new universe and interacting with the outer universes:

- `universe(`$\vec{V}$`) with` q `in` p `end` executes the process p with a dedicated clock and queue of nodes q until the queue is empty, the process "pauses up", stops or terminates. Every variable $st\ x \in \vec{V}$, where st is the spacetime of the variable, is transferred from the current layer to the declared universe, we call it a *top-down transfer*. When the queue q is not specified, the universe cannot manipulates `world_line` variables and cannot push nodes onto the queue.
- `pause up` suspends the current universe and gives the control back to the outer universe.
- `stop` suspends the execution of the current universe in the outermost universe—which is the environment.
- q`::`x is an expression for retrieving the value of the variable x in the queue q. We call it a *bottom-up transfer* because we can observe the value of a variable as if we were in the universe of this variable. We give an example of this expression in Section 3.6.2.

To illustrate this new statement, we consider the following example:

Example 3.3 (Single space universe)**.** We can execute a process on a different time scale which goes faster than the global clock. In spacetime, this faster clock is implemented with a universe encapsulating a code executed on a new time scale.

```
single_space LMax x = 0;
par
|| universe(single space x)           (A)
     x <- 1;
     pause;
     x <- 2;
     pause up;
     x <- 3
   end
|| System.out.println(read x);        (B)
   pause;
   System.out.println(read x);
end
```

There are two processes A and B increasing and reading the information in the variable x defined over the lattice $LMax$. The process B witnesses the write events $x \leftarrow 1$ and $x \leftarrow 2$ as simultaneous, and therefore prints the value 2 in its first instant. In contrast, the process A is executed on a faster time scale and witnesses these two events as separated. During the first instant of B, we executed two instants in A. Thereafter, the process A is suspended in its parent clock thanks to the statement `pause up`. In the second instant of B, and the third instant of A, A writes the value 3 in x, the universe of the process A terminates and the value 3 is printed by B. ⌟

3.5.2 Communication across layers of the hierarchy

A key characteristics of universe compositionality is to share data among parallel universes. Parallel universes communicate through variables declared in a common upper universe. The problem is to allow a variable to traverse layers such that every universe has a common reference to this variable. In the following, we refer to the innermost universe as U_4 in reference to the lattice L_4—in the case of spacetime, the lattice L_3 is treated by external functions (such as for propagation). In general, we call a universe with U_i where i refers to its level in the hierarchy. Before illustrating this notion with examples, we explain our notation to represent graphically a hierarchy of universes.

Notation 3.1 (Spacetime diagram)**.** Spacetime diagrams such as the one shown in Figure 3.4 must be interpreted as follows:

- A line between two dots represents a time instant where the first dot is the beginning of the instant and the second is the end of the instant.
- A box represents a set of synchronized universes with local time steps.
- For world lines a grey dot is a node that is added onto the queue but not instantiated yet.
- A white dot is a node that has already been instantiated in a former instant of the parent universe.

The following example illustrates how two universes communicate.

Example 3.4 (Communication in parallel universes)**.** We implement a process `pause_up_every` that pauses in the upper universe at regular intervals. For this purpose, we combine two universes where one counts the number of nodes and another pauses in the parent universe every `step` nodes.

```
single_space LMax N;
single_space LMax step;

proc pause_up_every =
```

```
par
|| universe(single_space N) in
     flow write N.inc() end
   end
|| universe(single_space N, single_space step) in
    loop
      when mod(read N, read step) == 0 then pause up
      else pause end
    end
   end
end
```

The universes are executed synchronously and communicate over the variable `N`. Note also that the universes are both without a queue and thus cannot contain statements relevant to the search tree. ⌟

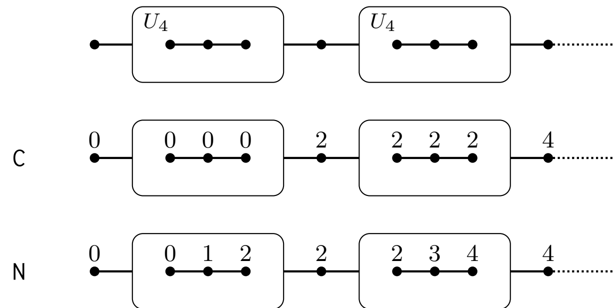


Figure 3.4: Variable transfer: `single_space` to `single_space` in `single_space` queue.

We show in the latest example that a variable `N` can traverse a layer whilst it is viewed with the same spacetime in both universes. Consider the following process:

```
single_space LMax N;
single_space LMax step;
single_space LMax C;
proc observe_l5 =
  N <- 0; step <- 2;
  par
  || pause_up_every()
  || flow C <- N end
  end
```

It executes the former process with the variable `step` sets to 2. The variables are transferred from a `single_space` declaration in U_5 to a `single_space` declaration in U_4. The result of the execution is shown in Figure 3.4. We observe that the universe U_4 executes two instants per instant of its parent's universe.

In this example, it is intuitive that if a variable evolves monotonically in U_4, then it also evolves monotonically in U_5. However, the `single_space` transfer is only one possible variable transfer among $3 \times 3 \times 3$ possibilities. Indeed, a

variable has a spacetime in U_i and can be seen with another spacetime in U_{i-1}. In addition, the transfer is operated through a universe with a queue in its own spacetime. However, variables cannot freely modify their spacetime when traversing a universe, because every transfer does not preserve the monotonicity of the computation (see Section 5.4.2 for the valid transfers).

3.6 Programming in L_5 with universes

We explain in more depth the notion of universe throughout two search strategies: iterative deepening search and branch and bound with universes.

3.6.1 Iterative deepening search

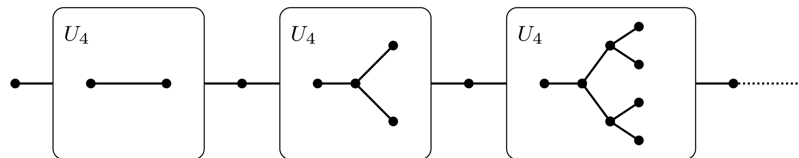


Figure 3.5: IDS computes over a `single_time` queue in every instant of the universe U_5.

We consider iterative deepening search (IDS) [Kor85] which explores the search tree at some depth and restarts the search on some depth limits (see Example 1.10). This strategy is defined on top of a `single_time` queue.

```
class IDS {
  single_time StackLR queue = new StackLR();
  single_space VStore domains;
  single_space CStore constraints;
  single_space LMax limit = new LMax(0);

  public IDS(VStore domains, CStore constraints) { ... }

  public proc search = par ids() <> solve() end

  flow solve =
    universe(world_line domains, world_line constraints) with queue in
      single_space Solver solver = new Solver(domains, constraints);
      solver.one_solution()
    end

  flow ids =
    readwrite limit.inc();
    universe(single_space limit) with queue in
      world_line LMax depth = new Depth(0);
      flow
        readwrite depth.inc();
```

```
          when limit |< depth then prune end
        end
      end
  }
```

We decompose this strategy into two main processes:

- `solve` is encapsulating CSP solving inside a universe. It instantiates the class `Solver` that we defined in Section 3.4.1 in order to find the first solution of a CSP.
- `ids` is counting the depth of the current tree, and when we reach some limit, we prune the subtree of the current node. In every instant of the parent universe, we increase by one the limit in order to explore a larger search tree.

We compose these two search strategies in the process `search` with the conjunctive parallel operator `<>`. An important compositionality aspect is that the semantics of `<>` is propagated inside the universes in order to compose their subtrees. Hence, the subtree generated by `solve` is discarded by the statement `prune` in the process `ids`.

IDS is illustrated on the spacetime diagram depicted in Figure 3.5. We observe that the exploration of the search tree is totally encapsulated inside a universe. The diagram does not represent the intermediate solving steps inside the universe U_4 but just the search tree explored during a step—this is why there is no white dot.

To conclude this example, the process `ids` is defined independently to the CSP exploration, and thus it could be re-used with another solving process. Creating a library of reusable and compositional search strategies is discussed in Chapter 6.

3.6.2 Branch and bound revisited

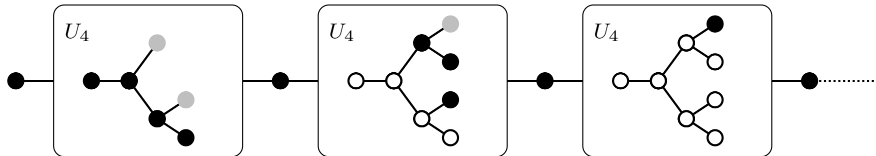


Figure 3.6: Pausing and resuming branch and bound exploration at solution nodes.

Complications arise in the semantics when considering world line variables in a single space universe (i.e. universe that contains `pause up` or `stop` statements). To understand the issue, consider the Figure 3.6 where the search is paused up when we reach a solution node. A variable is seen from two different perspectives: as a single space variable in U_5 and as a world line variable in U_4. These two views seem incompatible since U_5 can observe such a variable to evolve non-monotonically. For example, if we pause up in each solution node in U_4, the values observed by U_5 evolve non-monotonically since two solutions are not ordered.

An important aspect to the solution of this problem is that any world line variable is attached to a queue of nodes.

Every world line variable has a location in the queue of the current universe.

Therefore, whenever the spacetime of a variable is transferred to `world_line`, any value written inside this variable is bounded to the queue of the current universe. If the variable is also a `world_line` in the upper universe, then it also has a location inside the queue of this universe.

Writes on a world line variable are not propagated to its upper layers.

Therefore, any change on a transferred `world_line` variable is only observable in the lower levels. For example, it means that if we declare our CSP model in U_5 and solve it in U_4, the model in U_5 is left unchanged. An immediate question is how to retrieve the changes operated on a variable in a lower universe? This is the purpose of the bottom-up transfer $q::v$ where we can retrieve the value of v attached to a queue q. Moreover, writing into this variable in U_5 has the effect to write this information in the root node of the tree, and consequently in all the nodes of the queue in U_4. This transfer is useful for algorithms such as branch and bound: adding a constraint in the U_5-view of the variable updates the bound of the entire search tree in U_4. To make things more concrete, we give an example of this behavior.

Example 3.5 (Branch and bound revisited)**.** Consider the following branch and bound strategy maximizing the value of a variable `x`.

```
class MaximizeBAB {
  world_line VStore domains;
  world_line CStore constraints;
  single_space IntVar x;
  single_time LMax obj;

  public MaximizeBAB(VStore domains, CStore constraints,
    IntVar x) { ... }

  public proc search(q) =
    par
    || maximize(q)
    || yield_objective()
    end

  proc maximize(q) =
    universe(world_line domains, world_line constraints, single_space obj) with q in
      flow
        single_time ES consistent = read constraints.consistent(read domains);
        when consistent == true then
          obj <- x.getValue();
          pause up
```

```
        end
      end
    end

  flow yield_objective = constraints <- read x.gt(read obj);
}
```

The universe U_4 pauses up each time it reaches a solution, and it updates *obj* with the current value of the objective variable. It leads to a tree exploration similar to the one shown in Figure 3.6. The process `yield_objective` adds a new upper bound to the problem through the U_5 view of the variable `constraints`. This has the effect to constrain the remaining nodes in U_4 automatically. In practice, it is implemented by adding these constraints to the root node of the search tree so it stays active for the whole search. The advantage is that we do not need to manage the version of the objective constraint as in Example 3.4.2.

We combine this branch and bound algorithm with a process that prints every intermediate solution.

```
proc bab_with_print(domains, constraints, x) =
  single_space StackLR q = new StackLR();
  single_space MaximizeBAB bab = new MaximizeBAB(domains, constraints, x);
  par
  || bab.search(q);
  || flow System.out.println("Solution: " + read q::domains) end
  end
```

The expression `q::domains` allows us to access to the value of domains as observable in the universe with the queue q. In contrast, the variable `domains` only contains the initial problem without the changes made in the lower layers. Besides, we can define other universes working on the queue `q`. They would be synchronized in time and space with this branch and bound strategy. ⌟

Another example where variable transfer is useful is for dynamically creating a CSP. In the examples above, we see the universe U_5 as a single space universe: it does not generate a search tree in U_5. However, it is sometimes useful to manipulate nested combinatorial structures. This can be obtained by `world_line` variables traversing `world_line` universe. Such a feature is useful to program a model dynamically generating another model. Chapter 8 is dedicated to such a strategy where a model (from model checking theory) dynamically generates a CSP.

Chapter 4

Behavioral Semantics Inside an Instant

This chapter formalizes the behavioral semantics of spacetime which extends the synchronous paradigm to lattice-based variables. Mainstream synchronous languages such as Esterel or Lustre are restricted to variables defined over flat lattices. Consequently, the value of a variable evolves only once during an instant—from the bottom element of its lattice to a concrete value. In contrast, with general lattices, a variable can evolve several times during one instant. This semantics generalizes the concurrent model of communication between processes. We can read and write more than one time on the same variable. To this end, we introduce a suited semantics that takes into account the scheduling of successive read and write operations within an instant. In summary, this chapter focusses on the semantics describing the evolution of the space—the set of variables of the program—during a single instant. We tackle the semantics of a process evolving across instants, and its associated statements, in Chapter 5.

4.1 Behavioral semantics

The semantics of spacetime is inspired by the logical behavioral semantics of Esterel as defined in [BG92, Ber02]. Behavioral semantics is a "big-step" semantics based on natural deduction. As mentioned in [Ber02], behavioral semantics is mathematically simpler than operational semantics, and thus more adequate to formalize properties such as reactivity and determinism. However, it is not appropriate for implementation purposes since this semantics does not describe *how* a reaction is computed but rather *what* is a valid reaction. Therefore, a derivation in the behavioral semantics is a proof that a program transition is valid.

Behavioral transition rule

The behavioral semantics consists in a set of transition rules specifying the reaction of the program within an instant. A behavioral transition rule takes the form:

$$N \vdash p \xrightarrow[S]{S', B, k} p'$$

where N is the set of names, p and p' are processes, S is the set of input and output variables, S' is the set of the output and local variables, B is the output set of branches and k is the completion code. Intuitively, it means that the program p reacts to the input variables in S by producing the branches B and the output variables in S'. The set S is a lattice called the space of the program and is defined in Section 4.1. The set of branches B allows us to compose the search trees generated by several processes (Section 4.2). During the reaction, the program p is rewritten into p' which has the completion status k (e.g. p' is terminated or paused). The set of names N provides fresh names when declaring variables, its purpose is to give a unique identifier to every variable—it comes from the semantics of ReactiveML [Man06]. It is similar to the indexed sets used in Chapter 1. We have the disjoint union of the set of names $N_1 \dot\cup N_2 = N$ such that $N_1 \cup N_2 = N$ and $N_1 \cap N_2 = \emptyset$.

We stress one more time the peculiarity of behavioral semantics before further defining the semantics of spacetime. Behavioral semantics is best thought as a formal logic system that do not compute but prove that a formula—a program's reaction in our case—is valid. To do so, we must have the input/output space S before we start to derive the proof tree. In fact, this semantics is useful to prove some properties: is there a proof of a program that does not react at all (reactivity property)? Can a reaction provide two different output spaces such that both have a valid proof tree (deterministic property)? Answering such questions is more easily done in the behavioral semantics than in an operational semantics.

There is another important aspect to keep in mind with behavioral transition. We *read* from S but we *write* in S'. Due to the invariant $S \models S'$, formalizing the notion of instantaneous, we can only read what has been written somewhere in the program (or from the user input). It formalizes the fact that two processes in parallel have access to the information of one another. This is because we know "in advance" what the other processes have written. The goal of the proof tree is to check that processes actually wrote the information during the instant, and did not write something else.

Completion code

At the end of an instant, a process can be either normally terminated (code 0), paused in a universe (code 1), paused in its parent's universe (code 2) or paused in the outermost universe (code 3). Accordingly, we assign a completion code to a process P as follows:

$$compl(P) = \begin{cases} 0 & \text{if } P \text{ is terminated} \\ 1 & \text{if } P \text{ is paused} \\ 2 & \text{if } P \text{ is paused up} \\ 3 & \text{if } P \text{ is stopped} \end{cases}$$

The completion code plays an important role to define the semantics rules since they act differently according to the state of the program.

Space of the program

The variable environment of a program, called its *space*, stores the spacetime attribute and the value of the variables.

Definition 4.1 (Spacetime attribute)**.** *The spacetime attributes are given by the set spacetime* $= \{\rightarrow, \circlearrowleft, \downarrow\}$ *where* $\rightarrow$ *stands for* `single_space`*,* $\circlearrowleft$ *stands for* `single_time` *and* $\downarrow$ *stands for* `world_line`[1]*.*

We do not consider the types of the lattices in the semantics and we delegate typing to the host language. Moreover, the set of the types of all the variables is known at the compilation, so we can define a type-erasure lattice using the disjoint union lattice derivation (Definition 1.19).

Definition 4.2 (Type-erasure)**.** *Given a collection of lattices* $(T_1, \ldots, T_n)$*, its type-erasure is given by the lattice* $Type = T_1 \mathbin{\dot\cup} \ldots \mathbin{\dot\cup} T_n$*.*

Therefore, any spacetime variable takes a value in the lattice *Type*. From this, we build the space of the program using the lattice derivation *Store* (Definition 1.25).

Definition 4.3 (Spacetime variable)**.** *The set of spacetime variables is a lattice given by the Cartesian composition of the set spacetime and the lattice type defined as follows:*

$$Var(Type) = \{\top\} \oplus (spacetime \times Type) \oplus \{\bot\}$$

We need a distinct top element $\top$ for representing variables that are merged with a different spacetime, and bottom element $\bot$ for variable without spacetime. The top element indicates that the spacetime of a variable cannot change during its lifetime.

Definition 4.4 (Space of the program)**.** *Given the type-erasure Type of a program and a set of locations Name, the space of the program is a lattice defined as follows:*

$$Space(Name, Type) = Store(Name, Var(Type))$$

The entailment is inherited from Store. Given a space S*, we define the subsets of the single space variables with* $S^{\rightarrow}$*, the single time variables with* $S^{\circlearrowleft}$ *and the world line variables with* $S^{\downarrow}$*.*

The value $v \in Type$ at the location x can be accessed with $S^V(x)$ if $(st, v) \in S(x)$. In addition, given an element $(st, v) \in S(x)$, we define the projections $S^{st}(x) \mapsto st$ and $S^V(x) \mapsto v$ to respectively extract the spacetime and the value of a variable x.

[1] These symbols reflect how the variables evolve in the search tree. For example, $\downarrow$ depicts an evolution from the root to a leaf of the tree along a path.

4.2 Composition of search trees

We now give a formal definition of the composition in time and space of the search trees generated by two processes. A process generates a search tree across the instants and a sequence of branches within an instant. In particular, the branches created with the statement `space` are composed differently depending on whether they are composed with a parallel or sequence operator. To define the composition of search trees, we use the following relevant subset of spacetime:

$\langle p,\ q\rangle ::= p\ \texttt{;}\ q \mid p\ \texttt{||}\ q \mid p\ \texttt{<>}\ q \mid \texttt{space}\ p \mid \texttt{prune} \mid \alpha$

where (i) $p, q \in P$ with P the set of all the processes, and (ii) α is an atomic statement which is not composed of other statements.

These statements form the relevant fragment of spacetime for composing branches. We can extend the definitions given below to the full spacetime language without compositional issues.

To start our compositional semantics, we first define an algebra of *branch processes* with algebraic operators to compose branches.

Definition 4.5 (Branch process)**.** *The set of all branch processes is defined as* $B = \{\texttt{space}\ p \mid p \in P\} \cup \{\texttt{prune}\}$*. That is, a branch is either labelled by a process* p *or pruned.*

Definition 4.6 (Branch algebra)**.** *The branch algebra is defined over a sequence of branches* $\langle B^n, \circ, \vee, \wedge\rangle$ *where the operators are defined below. The empty sequence* $\langle\rangle$ *is the identity element of these three operators.*

- $\circ$ *is a noncommutative and associative concatenation operator. It describes the search tree generated by the sequence operator* $p\ \texttt{;}\ q$*.*
- $\vee$ *is a commutative and associative disjunctive operator. The element* $\langle\texttt{prune}\rangle$ *is a second identity element for* $\vee$*. It relates to the disjunctive parallel operator* $p\ \texttt{||}\ q$*.*
- $\wedge$ *is a commutative and associative conjunctive operator. The element* $\langle\texttt{prune}\rangle$ *is an absorbing element for* $\wedge$*. It describes the conjunctive parallel operator* $p\ \texttt{<>}\ q$*.*

Given $+ \in \{\circ, \vee, \wedge\}$*, we have the following identity laws common to all three operators:*

$$\langle\rangle\ +\ \mathit{branches} = \mathit{branches}$$
$$\mathit{branches}\ +\ \langle\rangle = \mathit{branches}$$

Sequence composition Given $b_i, b_j \in B$ with $1 \leq i \leq n$ and $1 \leq j \leq m$, we define the sequence operator $\circ$ that performs the concatenation of two sets of branches as follows:

$$\langle b_1, \ldots, b_n\rangle \circ \langle b'_1, \ldots, b'_m\rangle = \langle\ b_1, \ldots, b_n, b'_1, \ldots, b'_m\ \rangle$$

Parallel compositions We define an operator $\vee^1$ and $\wedge^1$ to combine two branch processes, and we then lift these operators to sequence of branches. Two set of branches are combined by repeating the last element of the shortest sequence when the sizes differ. Given $p, q \in P$ and $b \in B$, we define the disjunctive parallel operator $\vee$ between two branches as follows:

$$
\begin{aligned}
b \vee^1 \mathsf{prune} &= b \\
\mathsf{space}\ p \vee^1 \mathsf{space}\ q &= \mathsf{space}\ (p\ \mathsf{||}\ q) \\
\langle b_1, \dots, b_n \rangle \vee \langle b'_1, \dots, b'_m \rangle &= \begin{cases} \langle\ b_1 \vee^1 b'_1,\ \dots,\ b_{n-1} \vee^1 b'_{m-1},\ b_n \vee^1 b'_m\ \rangle & \text{if } n = m \\ \langle\ b_1 \vee^1 b'_1,\ \dots,\ b_{n-1} \vee^1 b'_m,\ b_n \vee^1 b'_m\ \rangle & \text{if } n > m \end{cases}
\end{aligned}
$$

Similarly, we define the conjunctive parallel operator $\wedge$:

$$
\begin{aligned}
b \wedge^1 \mathsf{prune} &= \mathsf{prune} \\
\mathsf{space}\ p \wedge^1 \mathsf{space\ q} &= \mathsf{space}\ (p\ \mathsf{<>}\ q) \\
\langle b_1, \dots, b_n \rangle \wedge \langle b'_1, \dots, b'_m \rangle &= \begin{cases} \langle\ b_1 \wedge^1 b'_1,\ \dots,\ b_{n-1} \wedge^1 b'_{m-1},\ b_n \wedge^1 b'_m\ \rangle & \text{if } n = m \\ \langle\ b_1 \wedge^1 b'_1,\ \dots,\ b_{n-1} \wedge^1 b'_m,\ b_n \wedge^1 b'_m\ \rangle & \text{if } n > m \end{cases}
\end{aligned}
$$

Given a process p, we obtain its sequence of branches inductively as follows:

$$
\begin{aligned}
\mathit{branches}(\mathsf{space}\ p) &= \langle \mathsf{space}\ p \rangle \\
\mathit{branches}(\mathsf{prune}) &= \langle \mathsf{prune} \rangle \\
\mathit{branches}(\alpha) &= \langle \rangle \\
\mathit{branches}(p\ \mathsf{;}\ q) &= \begin{cases} \mathit{branches}(p) \circ \mathit{branches}(q) & \text{if } p \text{ is instantaneous.} \\ \mathit{branches}(p) & \text{if } p \text{ has reached a } \mathsf{pause} \text{ statement.} \end{cases} \\
\mathit{branches}(p\ \mathsf{||}\ q) &= \mathit{branches}(p) \vee \mathit{branches}(q) \\
\mathit{branches}(p\ \mathsf{<>}\ q) &= \mathit{branches}(p) \wedge \mathit{branches}(q)
\end{aligned}
$$

Although this function branch is not used in the semantics, it provides a clear summary of how the branches are composed as shown in the following example.

Example 4.1 (Composition of branches)**.** We illustrate this semantics with several examples of processes along with the sequence of branches they generate. We create a N-ary trees with the sequence operator:

$$\mathit{branches}(\mathsf{space}\ p\ \ \mathsf{;}\ \ \mathsf{space}\ q\ \ \mathsf{;}\ \ \mathsf{space}\ r) = \langle \mathsf{space}\ p, \mathsf{space}\ q, \mathsf{space}\ r \rangle$$

If we replace the sequence with the disjunctive parallel, we obtain a singleton branch:

$$\mathit{branches}(\mathsf{space}\ p\ \mathsf{||}\ \mathsf{space}\ q\ \mathsf{||}\ \mathsf{space}\ r) = \langle \mathsf{space}\ (p\ \mathsf{||}\ q\ \mathsf{||}\ r) \rangle$$

We prune the whole subtree of a process $p \in P$ with:

$$\mathit{branches}(p\ \mathsf{<>}\ \mathsf{prune}) = \langle \mathsf{prune} \rangle$$

and we prune all the branches but the left one with:

$$branches(\texttt{(space nothing) ; prune}) = \langle \texttt{space nothing}, \texttt{prune} \rangle$$

Finally, consider the two processes:

$$\begin{aligned} branches(\texttt{(space } p \texttt{ ; prune) <> (prune ; nothing)}) &= \langle \texttt{prune} \rangle \\ branches(\texttt{(space } p \texttt{ ; prune) || (prune ; nothing)}) &= \langle \texttt{space } p, \texttt{prune} \rangle \end{aligned}$$

In the first case, the sequence of branches generated by the right side is $\langle \texttt{prune} \rangle$ because `nothing` alone generates the neutral element $\langle\rangle$ which is absorbed by the sequence operator. In the second case, we keep `space` p because we perform the union of the branches. ⌟

Design rational 2: Coherence between space and time

The compositional semantics of the operators `||` and `<>` is not hazardous. It preserves the coherency between space and time. In the time dimension, the operator p `||` q terminates only once p and q have terminated while p `<>` q terminates once p or q has terminated. We preserve an identical semantics for the spatial dimension. We compose two the search trees by union with p `||` q and by intersection with p `<>` q. This is why we propose these two operators instead of combining every possibility and have four operators.

4.3 Expressions and interface with host

In this section, we define the behavioral semantics rules for the expression fragment of spacetime. The expressions of spacetime are kept to a minimal set and we rely on the host language for the arithmetic and other computation. We first explain the expression rules for spacetime, and then we tackle the interactions with the host language. In particular, we require the host computation to preserve the monotonicity of the space of the program.

The main point of this section is to establish how the spacetime semantics and the host semantics interact. It is solved by defining a dedicated host transition rule:

$$e \xrightarrow[H]{H'}\!\!\!\!\twoheadrightarrow v$$

which reduces the expression e into the value v with the input variables in the host environment H and produces the output environment H'. To propose a semantics agnostic to the host language, we rely on a host environment H and on a pair of functions $(host, space)$ that respectively transform the space S into the host environment H and vice versa. We write $e \twoheadrightarrow v$ when the host is not supposed to

modify the space of the program. In addition, we annotate with e^h the counterpart of a spacetime expression e into the host language.

On the side of the spacetime expressions, we refine the behavioral transition to eliminate branches and completion codes because expressions terminate instantaneously:

$$e \xhookrightarrow[S]{S'} v$$

where the expression e is instantaneously reduced into the value v under the input/output space S, and it produces the output space S'. We now define the semantics rule of the spacetime's expressions.

The rule CONSTANT factors out all the rules for the ground expressions `bot`, `top`, `false`, `true`, `unknown` as well as the host constants. The logical operators `or`, `and` and `not` are defined in the host language, which is assumed to define them according to the Kleene truth table (Figure 3.1). We also leave the computation of the entailment on a lattice to the host language in the rule ENTAILMENT. These rules are defined as follows:

$$\textsc{constant}\ \frac{c^h \twoheadrightarrow c'}{c \xhookrightarrow[S]{\bot} c'} \qquad \textsc{or}\ \frac{e_1 \xhookrightarrow[S]{S'} v_1 \quad e_2 \xhookrightarrow[S]{S''} v_2 \quad v_1 \vee^h v_2 \twoheadrightarrow v'}{e_1\ \texttt{or}\ e_2 \xhookrightarrow[S]{S' \sqcup S''} v'}$$

$$\textsc{not}\ \frac{e \xhookrightarrow[S]{S'} v \quad \neg^h v \twoheadrightarrow v'}{\texttt{not}\ e \xhookrightarrow[S]{S'} v'} \qquad \textsc{and}\ \frac{e_1 \xhookrightarrow[S]{S'} v_1 \quad e_2 \xhookrightarrow[S]{S''} v_2 \quad v_1 \wedge^h v_2 \twoheadrightarrow v'}{e_1\ \texttt{and}\ e_2 \xhookrightarrow[S]{S' \sqcup S''} v'}$$

$$\textsc{entailment}\ \frac{e_1 \xhookrightarrow[S]{S'} v_1 \quad e_2 \xhookrightarrow[S]{S''} v_2 \quad v_1 \models^h v_2 \twoheadrightarrow v'}{e_1\ \texttt{|=}\ e_2 \xhookrightarrow[S]{S' \sqcup S''} v}$$

The host language must provide an implementation of the constants, logical operators which are written $\neg^h$, $\vee^h$ and $\wedge^h$, and the entailment $\models^h$.

The next rule reads the value of a variable x into the input space S. It materializes the fact that we do not compute value, but read into the space provided at the beginning of the derivation. The rule VAR is defined as:

$$\textsc{var}\ \ x \xhookrightarrow[S]{\bot} S^V(x)$$

We erase the access mode of a variable with the rule Access because it is only relevant to the causality analysis tackled in Section 4.5.

$$\frac{\text{ACCESS} \quad access \in \{\texttt{read}, \texttt{write}, \texttt{readwrite}\} \qquad x \xhookrightarrow[S]{\bot} v}{access\ x \xhookrightarrow[S]{\bot} v}$$

The main interaction with the host language is given by the rule HCALL which proves the well-formedness of a host function call:

$$\frac{\begin{array}{c}\text{HCALL} \\ \forall i \in [1..n],\ e_i \xhookrightarrow[S]{S'_i} v_i \text{ if } e_i \text{ is not a variable's access, otherwise } e_i \text{ is left unchanged} \\ f^h(v_1, \ldots, v_n) \xrightarrow[host(S)]{H'} v \qquad space(H') \models S\end{array}}{f(e_1, \ldots, e_n) \xhookrightarrow[S]{space(H') \sqcup \bigsqcup_{i \in [1..n]} S'_i} v}$$

We evaluate every parameter of the function but the variables because the host function reads and writes into the variables through its own host space $host(S)$. Moreover, in order to preserve the monotonicity of the computation, we verify that the host language writes into the space monotonically. To this purpose, we ensure that the space H' computed by the host transition evolves monotonically with the side condition $space(H') \models S$. To be totally correct, the host language should also verify that read-only variables passed as arguments are not written in the host function, and that write-only variables are not read in the host function. This is a challenging issue that is discussed more in Chapter 10 where we propose several directions to solve this problem.

4.4 Statements rules

The statements describe the temporal evolution of the space of the program. In the following sections, we focus on the communication, synchronous and search tree fragments of spacetime.

4.4.1 Communication fragment

The communication fragment of spacetime materializes the ask and tell metaphor of CCP (see Section 2.4.2). Firstly, the ask operation is implemented with the two rules WHEN-TRUE and WHEN-FALSE which evaluate the conditional expression e and execute one of the alternative of `when`:

WHEN-TRUE

$$\frac{e \overset{S'}{\underset{S}{\hookrightarrow}} true \qquad N \vdash p \xrightarrow[S]{S'',\, B,\, k} p'}{N \vdash \texttt{when}\ e\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \xrightarrow[S]{S' \sqcup S'',\, B,\, k} p'}$$

WHEN-FALSE

$$\frac{e \overset{S'}{\underset{S}{\hookrightarrow}} v \qquad v = false \lor v = unknown \qquad N \vdash q \xrightarrow[S]{S'',\, B,\, k} q'}{N \vdash \texttt{when}\ e\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \xrightarrow[S]{S' \sqcup S'',\, B,\, k} q'}$$

We implement CWA by evaluating *unknown* as $false$ in the rule WHEN-FALSE. It is valid because if e reduces to *unknown*, then this result is definitive in the current instant.

Next, the rule TELL evaluates e to the value v and performs the join of v and the input value retrieved from S:

TELL

$$\frac{e \overset{S'}{\underset{S}{\hookrightarrow}} v \qquad S''(x) = S(x) \sqcup (S^{st}(x), v)}{N \vdash x\ \texttt{<-}\ e \xrightarrow[S]{S' \sqcup S'',\, \langle\rangle,\, 0} \texttt{nothing}}$$

We allow the user to call host functions as statements, and to discard the result, if any. The rule CALL bridges between the call statement and the call expression:

CALL

$$\frac{f(e_1, \ldots, e_n) \overset{S'}{\underset{S}{\hookrightarrow}} v}{\{\} \vdash f(e_1, \ldots, e_n) \xrightarrow[S]{S',\, \langle\rangle,\, 0} \texttt{nothing}}$$

The last statement in the communication fragment is the declaration of a variable with the rule VAR-DECL:

VAR-DECL

$$\frac{N \vdash p[x \to n] \xrightarrow[S]{S',\, B,\, k} p' \qquad S'' = \{(n, (st, \bot))\}}{N \,\dot{\cup}\, \{n\} \vdash st\ Type\ x\ \ ;\ p \xrightarrow[S]{S' \sqcup S'',\, B,\, k} st\ Type\ x\ \ ;\ p'}$$

It extracts a fresh name n from the names' set N and substitutes x for n in the program p which is written $p[x \to n]$[2]. The most important aspect of this rule is

[2]The variable declaration must be evaluated with regards to its body, this is why the body p follows the declaration. In order to be coherent with the syntax, we can transform any variable declaration *st Type x* which is not followed by any statement to *st Type x*; `nothing`.

to substitute syntactic variables for their locations in the space. Hence if a variable x is allocated at the location 1 in the store, we substitute x for 1 in the program p. The substitution function is defined inductively over the structure of the program p. We give its two most important rules:

$$y[x \to n] \mapsto \begin{cases} n & \text{if } x = y \\ y & \text{if } x \neq y \end{cases}$$

$$(st\ Type\ y\ ;\ p)[x \to n] \mapsto \begin{cases} st\ Type\ y\ ;\ p & \text{if } x = y \\ st\ Type\ y\ ;\ p[x \to n] & \text{if } x \neq y \end{cases}$$

It replaces any identifier equals to x by n unless we went through a variable declaration with the same name.

4.4.2 Synchronous fragment

In this section, we adapt the synchronous statements of Esterel to the space dimension of spacetime. We start with the axioms rules NOTHING and PAUSE:

NOTHING
$$\{\} \vdash \texttt{nothing} \xrightarrow[S]{\bot, \langle\rangle, 0} \texttt{nothing}$$

PAUSE
$$\{\} \vdash \texttt{pause} \xrightarrow[S]{\bot, \langle\rangle, 1} \texttt{nothing}$$

We set the completion code to 0 and 1 which indicate that the statements are respectively terminated and paused. We leave the output space and the branch set empty.

The rule LOOP guarantees that p is not instantaneous by forbidding the completion code k to be equal to 0:

LOOP
$$\frac{N \vdash p \xrightarrow[S]{S', B, k} p' \qquad k \neq 0}{N \vdash \texttt{loop}\ p\ \texttt{end} \xrightarrow[S]{S', B, k} p'\ \texttt{;}\ \texttt{loop}\ p\ \texttt{end}}$$

To simulate a loop, we extract the body p of the loop and execute it before the loop. In a future instant, we will reach the statement `loop` again to execute the next iteration.

The next three statements are central to the composition of two processes in time and space. We compose the search trees created by distinct processes with the rules CONJ-PAR, DIS-PAR and SEQ-NEXT. These rules are built on the compositional semantics introduced in Section 4.2.

DIS-PAR

$$\frac{N_1 \vdash p \xrightarrow[S]{S',B,k} p' \qquad N_2 \vdash q \xrightarrow[S]{S'',B',k'} q'}{N_1 \dot{\cup} N_2 \vdash \texttt{par}\ p\ \texttt{||}\ q\ \texttt{end} \xrightarrow[S]{S' \sqcup S'', (B \vee B'), max^{\vee}(k,k')} \texttt{par}\ p'\ \texttt{||}\ q'\ \texttt{end}}$$

CONJ-PAR

$$\frac{N_1 \vdash p \xrightarrow[S]{S',B,k} p' \qquad N_2 \vdash q \xrightarrow[S]{S'',B',k'} q'}{N_1 \dot{\cup} N_2 \vdash \texttt{par}\ p\ \texttt{<>}\ q\ \texttt{end} \xrightarrow[S]{S' \sqcup S'', (B \wedge B'), max^{\wedge}(k,k')} \texttt{par}\ p'\ \texttt{<>}\ q'\ \texttt{end}}$$

The rules DIS-PAR and CONJ-PAR terminate when respectively both and one of their sub-processes have terminated. We compute the completion code of DIS-PAR and CONJ-PAR as:

$$max^{\vee}(k,k') = \begin{cases} k & \text{if } k \geq k' \\ k' & \text{otherwise} \end{cases}$$
$$max^{\wedge}(k,k') = \begin{cases} 0 & \text{if } k = 0 \vee k' = 0 \\ max^{\vee}(k,k') & \text{otherwise} \end{cases}$$

For the disjunctive parallel, we compute the greatest completion code with $max^{\vee}$ in any case because we need to wait the termination of both processes. In the case of the conjunctive parallel, the function $max^{\wedge}$ maps to the termination code as soon as one sub-process terminates, and otherwise acts as $max^{\vee}$. We continue with the two rules for the sequence operator:

SEQ-FIRST

$$\frac{N \vdash p \xrightarrow[S]{S',B,k} p' \qquad k \neq 0}{N \vdash p\ ;\ q \xrightarrow[S]{S',B,k} p'\ ;\ q}$$

SEQ-NEXT

$$\frac{N_1 \vdash p \xrightarrow[S]{S',B,0} p' \qquad N_2 \vdash q \xrightarrow[S]{S'',B',k'} q'}{N_1 \dot{\cup} N_2 \vdash p\ ;\ q \xrightarrow[S]{S' \sqcup S'', (B \circ B'), k'} q'}$$

The sequence rule SEQ-FIRST executes the first process only because it did not terminate in the current instant. In the case of SEQ-NEXT, we execute both processes and concatenate their subtrees with the operator $\circ$ (Section 4.2).

The temporal dimension of a spacetime program can be controlled with the statements `abort` and `suspend`. As for the statement `when`, the rules for `abort` and `suspend` evaluate a condition which can result in *true*, $false$ or *unknown*. When the condition is true, `abort` terminates immediately without executing the process p. In contrast, `suspend` pauses in the current instant by setting its completion code to 1.

ABORT-TRUE

$$\frac{e \underset{S}{\overset{S'}{\hookrightarrow}} true}{\{\} \vdash \texttt{abort when } e \texttt{ in } p \texttt{ end} \xrightarrow[S]{S',\, \langle\rangle,\, 0} \texttt{nothing}}$$

ABORT-FALSE

$$\frac{e \underset{S}{\overset{S'}{\hookrightarrow}} v \qquad v = false \lor v = unknown \qquad N \vdash p \xrightarrow[S]{S'',\, B,\, k} p'}{N \vdash \texttt{abort when } e \texttt{ in } p \texttt{ end} \xrightarrow[S]{S' \sqcup S'',\, B,\, k} \texttt{abort when } e \texttt{ in } p' \texttt{ end}}$$

SUSPEND-TRUE

$$\frac{e \underset{S}{\overset{S'}{\hookrightarrow}} true}{\{\} \vdash \texttt{suspend when } e \texttt{ in } p \texttt{ end} \xrightarrow[S]{S',\, \langle\rangle,\, 1} \texttt{suspend when } e \texttt{ in } p \texttt{ end}}$$

SUSPEND-FALSE

$$\frac{e \underset{S}{\overset{S'}{\hookrightarrow}} v \qquad v = false \lor v = unknown \qquad N \vdash p \xrightarrow[S]{S'',\, B,\, k} p'}{N \vdash \texttt{suspend when } e \texttt{ in } p \texttt{ end} \xrightarrow[S]{S' \sqcup S'',\, B,\, k} \texttt{suspend when } e \texttt{ in } p' \texttt{ end}}$$

The rule FOR allows programmers to compose dynamically N processes where N is the runtime size of the range expression e. Hence, the process p is duplicated N times and expanded into $st\ Type\ x$; x <- v ; p to set the value of the variable x according to the current iteration. Importantly, the process p must be instantaneous otherwise the size of the program could grow indefinitely over time—a property we wish to avoid in order to keep the static analysis decidable and simple.

FOR

$$\frac{\begin{array}{c} e \underset{S}{\overset{S'}{\hookrightarrow}} e' \qquad N \vdash p' \xrightarrow[S]{S'',\, B,\, 0} p'' \\ p' = \Box\{(st\ Type\ x\ \texttt{;}\ x\ \texttt{<-}\ v\ \texttt{;}\ p) \mid v \in^h e'\} \qquad \Box \in \{\texttt{;}, \texttt{||}, \texttt{<>}\} \end{array}}{N \vdash \texttt{for } (\Box)\ (st\ Type\ x\ \texttt{:}\ e)\ p \texttt{ end} \xrightarrow[S]{S' \sqcup S'',\, B,\, 0} p''}$$

We iterate over the host range with $\in^h$ which means that the inclusion is a function from the host language. For each value in the range we create a set of processes P that are combined with $\Box\ P \stackrel{\text{def}}{=} p_1 \Box \ldots \Box p_n$ and $\Box$ is a combinator.

4.4.3 Search tree fragment

The search tree fragment is compact: we create or discard branches of the current node (see Section 4.2). We create a branch of the search tree in the rule SPACE

by adding `space p` into the set of branches. The rule PRUNE creates the sequence $\langle$`prune`$\rangle$ for discarding one or more branches in the current instant.

$$\textsc{space}\quad \texttt{space}\ p\ \texttt{end} \xrightarrow[S]{\bot, \langle \texttt{space}\ p \rangle, 0} \texttt{nothing} \qquad \textsc{prune}\quad \texttt{prune} \xrightarrow[S]{\bot, \langle \texttt{prune} \rangle, 0} \texttt{nothing}$$

4.5 Causality analysis

A program is causal (i.e. logically correct) if it is reactive and deterministic (Section 4.5.1). In spacetime, the causality analysis boils down to an analysis that ensures we can sequentialize the parallel branches of a program. A spacetime program can be sequentialized if it has an ordering of the `read`, `readwrite` and `write` accesses on variables such that the space evolves monotonically. To this purpose, we propose a constraint model of a spacetime program modelling the dependencies between these accesses (Section 4.5.2). Therefore, the program is causal if the constraint model generated is satisfiable. We give a function to build this constraint model of causality in Sections 4.5.4 and 4.5.5.

4.5.1 Logical correctness

Logical correctness is a major semantics aspect in synchronous languages [Ber02]. A spacetime program p is logically correct if it is reactive (at least one fixed point) and deterministic (at most one fixed point) for any input space S. We illustrate these two properties with standard examples adapted from Esterel [Ber02].

A spacetime program p is deterministic if it has at most one output space. For example, consider the following non-deterministic program:

```
single_time LMax x = new LMax(0);
when x |= 1 then x <- 1 end
```

We can prove this program with two different input/output spaces: $S_1 = \{(a, (\circlearrowleft, 0))\}$ and $S_2 = \{(a, (\circlearrowleft, 1))\}$ where a is the location of x in the space. To illustrate these possibilities, we give the two corresponding derivations. For clarity, we simplified the declaration of x to $x^{\circlearrowleft}$ `= 0`.

$$\textsc{var-decl}\ \dfrac{\textsc{when-false}\ \dfrac{\textsc{entailment}\ \dfrac{\textsc{var}\ \dfrac{S_1^V(\mathsf{a}) = 0}{\mathsf{a} \overset{\bot}{\underset{S_1}{\hookrightarrow}} 0} \qquad 1 \overset{\bot}{\underset{S_1}{\hookrightarrow}} 1 \qquad 0 \models^h 1 \twoheadrightarrow \mathit{unknown}}{\mathsf{a}\ \texttt{|=}\ 1 \overset{\bot}{\underset{S_1}{\hookrightarrow}} \mathit{unknown}}}{\{\} \vdash \texttt{when a |= 1 then a <- 1 end} \xrightarrow[S_1]{\bot, \langle\rangle, 0} \texttt{nothing}}}{\{a\} \vdash x^{\circlearrowleft}\ \texttt{= 0 ; when}\ x\ \texttt{|= 1 then}\ x\ \texttt{<- 1 end} \xrightarrow[S_1]{\{(a,(\circlearrowleft,0))\}, \langle\rangle, 0} \texttt{nothing}}$$

The second derivation reacts in the space S_2. We omit the derivation of the constant 1 in the rule ENTAILMENT for clarity purpose.

$$\dfrac{\dfrac{\dfrac{\dfrac{S_2^V(\mathtt{a}) = 1}{\mathtt{a} \overset{\bot}{\underset{S_2}{\hookrightarrow}} 1}\,\text{VAR} \qquad 1 \models^h 1 \twoheadrightarrow \mathit{true}}{\mathtt{a\ |=\ 1} \overset{\bot}{\underset{S_2}{\hookrightarrow}} \mathit{true}}\,\text{ENTAILMENT} \qquad \dfrac{1 \overset{\bot}{\underset{S_2}{\hookrightarrow}} 1}{\mathtt{a\ <\!-\ 1} \xrightarrow[S_2]{\{(a,(\circlearrowleft,1))\},\,\langle\rangle,\,0} \mathtt{nothing}}\,\text{TELL}}{\{\} \vdash \mathtt{when\ a\ |=\ 1\ then\ a\ <\!-\ 1\ end} \xrightarrow[S_2]{\{(a,(\circlearrowleft,1))\},\,\langle\rangle,\,0} \mathtt{nothing}}\,\text{WHEN-TRUE}}{\{a\} \vdash x^{\circlearrowleft}\ \mathtt{=\ 0\ ;\ when}\ x\ \mathtt{|=\ 1\ then}\ x\ \mathtt{<\!-\ 1\ end} \xrightarrow[S_2]{\{(a,(\circlearrowleft,1))\},\,\langle\rangle,\,0} \mathtt{nothing}}\,\text{VAR-DECL}$$

For both input/output spaces S_1 and S_2, we obtain a valid proof tree of the program.

Secondly, we say a spacetime program p is reactive if it has at least one output space. The following program cannot react in any possible input/output space:

```
single_time LMax x = new LMax(0);
when x |= 1 then nothing else x <- 1 end
```

If the variable x takes the value 0, then the else-branch is executed and x must be equal to 1 which contradicts the entailment condition.

Therefore, our behavioral semantics is incomplete. We must reject programs that are non-deterministic and non-reactive, and we introduce the causality analysis for this purpose in the next section.

4.5.2 A constraint model of causality

We propose to model causality with a constraint model which has the advantage of keeping the analysis separated from the semantics rules. Our proposed causality analysis is incomplete because we do not take into account the universes; this is left for future work.

The causal dependencies in the program are generated by interleaving read and write operations on the space of the program. Our approach is to generate constraints to ensure that the program is *sequentially consistent* and monotonic. Sequential consistency ensures that the processes can be statically interleaved and the program executed in a sequential fashion. Monotonicity means that every read on a variable happens after the writes on the same variable. To verify these two properties, we propose a model written in MiniZinc (see Section 1.1.1). We comment every part of the model, and we start with the parameters of the model.

```
int: operations;
int: vars;
enum Access = { write, readwrite, read };
```

Our model is parametrized by the number of operations (`operations`) and the number of variables (`vars`) occurring in the current instant of the program. An operation is an access to a variable such as with `read` x. For clarity, we represent each access mode with an associated constant parameter.

```
set of int: Op = 1..operations;
set of int: Order = 1..operations;
set of int: Vars = 1..vars;

array[Op] of Vars: var_of_op;
array[Op] of Access: access_of_op;
array[Op] of bool: activated;
```

In addition to the number of operations and variables, we also parametrize the model with three arrays modelling three attributes of an operation:

- `var_of_op[`*op*`] =` v maps an operation indexed *op* to the variable v involved in the access operation. For example, if `read n` is the operation indexed 3, then we have `var_of_op[3] = n`.
- `access_of_op[`*op*`] =` a maps an operation to its access mode. With `read n`, we have `access_of_op[3] = read` where `read` is the constant given above.
- `activated[`*op*`] =` b enforces sequential constraints to be activated or not on the operation *op*.

The latest array `activated` deserves attention. We provide every possible operation of the program to the model, but a model only analyses a single path in the program. Therefore, if we have the process `when x |= y then x <- 2 else y <- 2`, only one of the processes `x <- 2` and `y <- 2` is activated during an execution.

These three arrays are given by the function building the causality model below. From this model, we schedule the program by finding a total order between the access operations, if it exists. The ordering is stored in the following array:

```
array[Op] of var Order: order_of_op;
```

Given `order_of_op[`*op*`] =` i, an operation *op* is scheduled at position i. If we invert this array—the indices becomes elements and vice versa—then we obtain the sequential order of the access operations.

Although the dependencies between variable accesses are generated in the next section, we can enforce three constraints that must always be fulfilled.

```
constraint all_different(order_of_op);
```

Firstly, the scheduling position of each operation must be different. Otherwise we could execute two operations at the same time and our execution would not be sequential. The next constraints are defined over each pair of activated operations on a same variable:

```
constraint forall(op1 in Op, op2 in Op where op1 != op2 /\
    var_of_op[op1] == var_of_op[op2] /\ activated[op1] /\ activated[op2])
    % (C1)
    (access_of_op[op1] > access_of_op[op2] -> order_of_op[op1] > order_of_op[op2]
    % (C2)
 /\ (access_of_op[op1] == readwrite /\ access_of_op[op2] == readwrite) -> false);
```

The first constraint `(C1)` enforces that every access `read` is done after `readwrite`, and in turn that every `write` is realized after a `readwrite`. The second constraint `(C2)` enforces that a variable is not accessed two times with `readwrite`.

To illustrate our constraint model, we give two examples of constraint model of causality.

Example 4.2 (Model of a non-deterministic process)**.** We first introduce a classic non-deterministic process where reads and writes on a same variable are inverted in two processes:

```
par x <- read y + 1 || y <- read x + 1 end
```

There are four operations with two reads and two writes, and two variables x and y. The MiniZinc model is given as follows:

```
operations = 4;
vars = 2;

var_of_op = [1, 2, 2, 1];
access_of_op = [write, read, write, read];
activated = [true, true, true, true];

constraint order_of_op[1] > order_of_op[2];
constraint order_of_op[3] > order_of_op[4];
```

The constraints generated ensures that the write on x happens after the read of y, and vice versa. We also activate all the operations since everything needs to be executed. It is immediately detected unsatisfiable by the constraint solver. ⌟

Example 4.3 (Model of a causal process)**.** The following process illustrates a `when` statement with valid *write after read*:

```
when x |= y then x <- 2 else y <- 3 end
```

In total, we have four operations and two variables, but only three operations can be active at any time. We gives a first model for the branch `then`:

```
operations = 4;
vars = 2;

var_of_op = [1, 2, 1, 2];
access_of_op = [read, read, write, write];

activated = [false, true, true, false];
constraint (order_of_op[2] < order_of_op[3]);
```

We deactivate the write operation on y since we do not go inside the branch `else`. The interesting part is that we also deactivate the sequential constraints of the read operation on x in the entailment. If we take the branch `then`, even if the variable x is written in later on, it cannot alter the result as long as y is fixed; this is why a write operation on x is allowed. More formally, since all the variables evolve monotonically, once y is fixed and $x \models y$ holds, for any new write v in x, we have $x \sqcup v \models y$. We apply a similar reasoning with the branch `else`:

```
activated = [true, false, false, true];
constraint (order_of_op[1] < order_of_op[4]);
```

In case $x \models y$ does not hold, we deactivate y because even if we write into y it will not change the result of the entailment.

To sum up, we obtained two constraint models, and both must be satisfiable in order to prove the causality of the process. ⌟

The goal of the next two sections is to generate the constraint parameters and sequential constraints of a spacetime process.

4.5.3 From the spacetime program to the causality model

We prepare a spacetime program before the causality analysis. The very first step is to index every spacetime operation with an integer *op* referring to its index in the arrays `var_of_op`, `access_of_op` and `order_of_op`. This can be realized statically by traversing the syntactic structure of the program. Every access operation `read n` is indexed with `read`^{op} `n` where *op* is the index of this operation. Variables with an implicit access mode, such as in `n <-` e, are directly indexed; for example with `n`^{op} `<-` e.

Once this indexing is done, we can fill the arrays `var_of_op` and `access_of_op` with the associated value, which corresponds to information obtained syntactically. For every operation that is not accessible in the current instant, we assign `false` in the corresponding index of the array `activated`. Therefore, by activating or not operations in the set `activated`, we can simulate that we are in a specific instant of the program.

The causality analysis is started with the program's operations indexed, the constraints given in Section 4.5.2, and the variables `operations`, `vars`, `var_of_op` and `access_of_op` fully defined. We perform the causality analysis on the program statements with the function *causal* which is defined in Section 4.5.5. However, the causal constraints are generated by the read and write on variables, which are situated inside an expression. The function *causal* is relying on the function *deps* for computing these dependencies, and thus we present *deps* first.

4.5.4 Causal dependencies

Causal dependencies of an expression are inductively computed with the function:

$$deps(e, m, M_3) \mapsto M_3'$$

where e is the expression, m is a boolean indicating if the expression is in a monotonic or anti-monotonic context, M_3 is the current causal model, and M_3' is the causal model updated with the dependencies introduced by e. The difference between a monotonic and non-monotonic context is that a monotonic context allows variables to be read now and written in later, while it is forbidden in a non-monotonic context. A causal model is a lattice structure

$$\begin{array}{l} M_3 : L_3 \times \mathcal{P}(\mathbb{N}) \times ES \\ (\langle d, P\rangle, Last, Inst) \in M_3 \end{array}$$

where $\langle d, P\rangle$ is the causal CSP, *Last* is the set of the latest read-write operations performed "just-before" the current expression or statement, and *Inst* indicates if the expression or statement is instantaneous or not. We use a projection of variables, written $d[O]$ for the array `order_of_op`, and $d[A]$ for the array `activated`.

We apply *deps* inductively on the boolean expressions and the arguments of a function call:

$$\begin{array}{ll} deps(\mathtt{c}, m, (\langle d, P\rangle, Last, Inst)) & \mapsto (\langle d, P\rangle, Last, true) \\ deps(\mathtt{not}\ e, m, M) & \mapsto deps(e, m, M) \\ deps(e_1\ \mathtt{and}\ e_2, m, M) \text{ or} & \\ deps(e_1\ \mathtt{or}\ e_2, m, M) & \mapsto deps(e_1, m, M) \sqcup deps(e_2, m, M) \\ deps(f(e_1, \ldots, e_n), false, M) & \mapsto \bigsqcup_{i \in [1..n]} deps(e_i, false, M) \end{array}$$

When we reach a constant c, we map to the same model with the instantaneous boolean sets to *true*. We notice that we only allow host functions in a non-monotonic context: once the arguments are passed to the function, they cannot gain more information anymore later. We enforce this condition because host function are only called once, and thus the read/write accesses must be ordered.

The three remaining atomic expressions are the accesses to the variables:

$$\begin{array}{l} deps(\mathtt{read}^{op}\ x, false, M) \text{ or} \\ deps(\mathtt{write}^{op}\ x, false, M) \text{ or} \\ deps(\mathtt{readwrite}^{op}\ x, false, M) \mapsto \\ \quad (\langle d \sqcup (d[A](op) \sqcup true), P \sqcup \bigsqcup_{\ell \in Last} [\![d[O](op) > d[O](\ell)]\!] \rangle, \{op\}, true) \\ \ \text{where } M = (\langle d, P\rangle, Last, Inst) \end{array}$$

$$\begin{array}{l} deps(\mathtt{read}^{op}\ x, true, M) \text{ or} \\ deps(\mathtt{write}^{op}\ x, true, M) \text{ or} \\ deps(\mathtt{readwrite}^{op}\ x, true, M) \mapsto M \end{array}$$

We set the activation status $d[A](op)$ of the operation op to *true* only when we are in a non-monotonic context. Then, we create one sequential constraints for each

last operation with $\bigsqcup_{\ell \in Last} [\![d[O](op) > d[O](\ell)]\!]$, and add these constraints into the CSP. Finally, we create a new set of latest operation with $\{op\}$, so that the next sequential operation is scheduled to happen after *op*. In a monotonic context, we do not add any constraint into M because further read and write do not impact the monotonicity of the computation.

The two last expressions are the entailment and tell operations:

$$\begin{array}{ll} deps(\texttt{n}^{op}\ \texttt{<-}\ e, false, M) & \mapsto deps(\texttt{write}^{op}\ \texttt{n}, false, M) \sqcup deps(e, false, M) \\ deps(e_1\ \texttt{|=}\ e_2, true, M) & \mapsto deps(e_1, true, M) \sqcup deps(e_2, false, M) \\ deps(e_1\ \texttt{|=}\ e_2, false, M) & \mapsto deps(e_1, false, M) \sqcup deps(e_2, true, M) \end{array}$$

We retrieve the dependencies of the tell operator by encapsulating the left variable `n` into a `write n` operation. Finally, we evaluate the left and right part of the entailment according to its context. The treatment of entailment will become clear when evaluating the conditional statement `when`.

4.5.5 Causal statements

Our causality analysis is defined inductively on the structure of the program with the following function:

$$causal(p, M_3, C) \to M_4$$

where p is the process to analyse, M_3 is the current constraint model, C is the continuation of the model construction, and M_4 is the L_4 derivation of M_3. The structure M_4 contains all the models that must be satisfiable. We need this structure to be at the fourth level of the lattice hierarchy because we need to collect disjunctive models when traversing the branches of the statement `when`.

We start with the causality analysis of the atomic statements of spacetime:

$$\begin{array}{l} causal(\texttt{pause}, M, C) \text{ or} \\ causal(\texttt{pause up}, M, C) \text{ or} \\ causal(\texttt{stop}, M, C) \mapsto \{M \sqcup_{M_3} (\bot, \{\}, false)\} \end{array}$$

$$\begin{array}{l} causal(\texttt{nothing}, M, C) \text{ or} \\ causal(\texttt{prune}, M, C) \text{ or} \\ causal(\texttt{space}\ p\ \texttt{end}, M, C) \mapsto C(M) \end{array}$$

On the one hand, the statements introducing a delay do not call the continuation C. They return the cumulated sequential model M with its instantaneous boolean value set to $false$. On the other hand, the next three atomic statements do not introduce any sequential dependency, and forward the creation of the model to the continuation.

The continuation C is created by the sequence statement:

$$causal(p \text{ ; } q, M, C) \mapsto causal(p, M, \lambda M'.causal(q, M', C))$$

We analyse p which is automatically followed by q when we hit the last instantaneous statement of p. If p contains a delay, the continuation will not be called.

The following statements describe the causality analysis of the ask and tell operations:

$$causal(st\ Type\ \mathtt{n}, M, C) \mapsto C(M)$$

$$causal(\mathtt{n}^{op} \texttt{ <- } e, M, C) \mapsto C(deps(\mathtt{n}^{op} \texttt{ <- } e, false, M))$$

$$causal(f(e_1, \ldots, e_n), M, C) \mapsto C(deps(f(e_1, \ldots, e_n), false, M))$$

$$\begin{aligned}&causal(\texttt{when } e \texttt{ then } p \texttt{ else } q \texttt{ end}, M, C) \mapsto \\ &\quad causal(p, deps(e, true, M), C) \sqcup_4 causal(q, deps(e, false, M), C)\end{aligned}$$

The three first statements generate the dependencies of their expressions, and call the continuation with the new causal model. We analyse the statement `when` by considering the entailment condition to be $true$ in one model and $false$ in the other model. We call the function *deps* on the expression e in a monotonic context if it is assumed to be $true$, and in a non-monotonic context if it is assumed to be $false$. The generated models are then merged in M_4 with the join operation $\sqcup_4$. We illustrate the causality analysis of the statements already introduced with the following example.

Example 4.4 (Sequence with possible pause). It is interesting to consider the model generated by the following process:

```
when x |= y then
  x <- 2
else
  y <- 3;
  pause
end;
x <- 3;
```

Of interest is the continuation $\lambda M.causal(\texttt{x <- 3}, M, C)$ created by the outermost sequence operator. In the first branch, we execute the continuation since the process is instantaneous. In the second branch, we hit the statement `pause`, and therefore we do not call the continuation. Without the `pause` statement in the branch `else`, we would call the continuation, and the causal model would be unsatisfiable because we write on x after an anti-monotonic read of x. ⌟

We consider next the two statements `abort` and `suspend` which are similar to the conditional statement:

$$causal(\texttt{abort when } e \texttt{ then } p \texttt{ end}, M, C) \mapsto$$
$$\quad causal(p, deps(e, false, M), C) \sqcup_4 C(deps(e, true, M))$$

$$causal(\texttt{suspend when } e \texttt{ then } p \texttt{ end}, M, C) \mapsto$$
$$\quad causal(p, deps(e, false, M), C) \sqcup_4 (deps(e, true, M) \sqcup_{M_3} (\bot_3, \{\}, false))$$

In case the expression e is assumed $false$, we evaluate the causal dependencies generated by the process p, and collect the non-monotonic dependencies generated by e. For the statement `abort`, when e is assumed $true$, we collect the monotonic dependencies of e and call the continuation C since the statement is terminated. For `suspend`, we also collect the monotonic dependencies of e, but we do not call the continuation C because `suspend` pauses when e is $true$. Therefore, we update the model with the instantaneous flag set to $false$.

The next statements are very close to their semantics rules. They are defined inductively by calling the access function over their statements and expressions. Of interest is the conjunctive parallel `<>` that merges the termination status with $\sqcup$ indicating that if one of the process is terminated, the whole parallel statement is terminated. This termination status is only used in the sequence statement p ; q where we analyse the process q only if p can terminate.

One of the most important part of the causality analysis is to deal with the parallel composition of two processes. We first generate the causal model of each branch independently and then combine the generated models with the following Cartesian product:

$$\begin{array}{ll} (\langle d, P\rangle, L, I) \times_3^+ (\langle d', P'\rangle, L', I') & \mapsto (\langle d, P\rangle \sqcup \langle d', P'\rangle, L \cup L', I + I') \\ M_4 \times_4^+ M_4' & \mapsto \{M_3 \times_3^+ M_3' \mid M_3 \in M_4 \wedge M_3' \in M_4'\} \end{array}$$

The operation $\times_4^+$ is parametrized by an operator $+$ used to combine the instantaneous flag of two models. It is the only distinction between two branches combined with `<>` and `||`, the first one is instantaneous if one of its two branches is instantaneous, whereas the second is instantaneous only if both branches are. We analyse the two parallel operators as follows:

$$causal(\texttt{par } p \texttt{ <> } q \texttt{ end}, M, C) \text{ or}$$
$$causal(\texttt{par } p \texttt{ || } q \texttt{ end}, M, C) \mapsto$$
$$\bigsqcup^4_{M_3 \in M_4} \begin{cases} C(M_3) & \text{if } M_3 \text{ is instantaneous} \\ \{M_3\} & \text{otherwise} \end{cases}$$
$$\text{where } M_4 = causal(p, M, \lambda M'.\{M'\}) \times_4 causal(q, M, \lambda M'.\{M'\})$$
with the operator $\times_4$ defined as $\times_4^{\sqcap}$ for `<>`, and as $\times_4^{\sqcup}$ for `||`

Once we created the Cartesian product of the models generated by both processes, we call the continuation C only on the models that are instantaneous. We illustrate this behavior on the following example.

Example 4.5 (Fork and possible merge)**.** The Cartesian product of two models might generate both instantaneous and delayed models, as shown in the following process:

```
par
|| x <- 2
|| when y |= x then y <- 3; pause end
end;
x <- 3;
```

The second branch pauses if we can deduce x from y, and otherwise the whole parallel statement terminates. Accordingly, the causality analysis generates two models where one is instantaneous and the other delayed. The continuation with the statement `x <- 3` is only called with the model in which the parallel statement terminates, and it is not called when it is delayed.

In short, the Cartesian product is necessary to combine models created by parallel processes, because the termination of such a process depends on some conditions. ⌟

The last two statements to analyse are the delayed loop and the instantaneous loop. The first one is defined as:

$$causal(\texttt{loop}\ p\ \texttt{end}, M, C) \mapsto causal(p, M, C)$$

It does not introduce dependency and forward the causality analysis to its body p.

The instantaneous loop `for` deserves some attentions since it generates at run-time some statements. We base our causality analysis on the following conjecture.

Conjecture 4.1 (Causality analysis of a `for` process)**.** *Given the statement*

$$\texttt{for}(\Box)(st\ Type\ x\ :\ e)\ p\ \texttt{end}$$

we duplicate the body $(n+1)$ *times where* n *is the number of execution paths of the process* p*. Then it is causal if the process*

$$(st\ Type\ x\ ;\ p)_1\ \Box \ldots \Box\ (st\ Type\ x\ ;\ p)_{n+1}$$

is causal.

That is, from the point of view of causality analysis, duplicating the process $n+1$ times is identical to duplicating the process an arbitrary number of times. We duplicate one more time to verify that a single execution path can be executed twice (for example to avoid `readwrite` two times on a same variable).

Assuming this conjecture is valid, then we can analyse a process `for` with:

$$causal(\mathtt{for(\Box)(}st\ Type\ x\ \mathtt{:}\ r\mathtt{)}\ p\ \mathtt{end}, M, C) \mapsto$$
$$causal(\mathtt{(}st\ Type\ \mathsf{a}_1\ \mathtt{;}\ p\mathtt{)}_1\ \Box \ldots \Box\ \mathtt{(}st\ Type\ \mathsf{a}_{n+1}\ \mathtt{;}\ p\mathtt{)}_{n+1}, M, C)$$

To prove this conjecture, we must first prove the base case where the composition of an arbitrary number of identical execution paths is causal only if the composition of two identical execution paths is causal. The reasoning can be extended to multiple execution paths, because there are a finite number of execution paths, and thus a finite number of their combinations.

4.6 Conclusion and discussion

We introduced the behavioral semantics of a process inside one instant. The main contribution of this chapter was to generalize the synchronous model of computation to general lattice, and to formalize the causality analysis of such processes. To this purpose, we took the approach of defining every structure manipulated in our semantics on lattice structures. Doing so, every semantics rule is a monotonic function over the space S, and therefore we guarantee by construction the determinism of the computation. However, it is necessary to provide a formal proof that every behavioral rule is indeed monotonic, and that our causality analysis always generates unsatisfiable models for non-deterministic and non-reactive processes. We do not provide such a treatment in this dissertation, but we believe that it is achievable with a theorem-prover because the structures underlying our semantics are well-defined.

To conclude this chapter, we discuss the differences between the causality analysis of Esterel and our work.

Potential analysis

The fragment of spacetime introduced in this chapter is built on Esterel. The differences almost all boil down to using variables defined over lattices instead of boolean values—the so-called signals in Esterel. It is useful to look at the semantic rules for declaring and testing the presence of a signal in Esterel as defined in Figure 4.1.

$Must_S$ and Can^+_S are called the "potential functions": they compute the set of signals that *must* and *can* be emitted during the current instant. In the rules $sig+$ and $sig-$, these functions are used to assign a value among $\{0, 1, \bot\}$ to the signal if it is absent, present or if it cannot be established yet. Thereafter, the conditional rules $present+$ and $present-$ are relying on the presence or absence of a signal in the environment S.

There is a very important distinction to be made between Esterel and spacetime. In Esterel, the data is syntactically rooted in the language, and thus deeply linked

$$\text{sig+}\ \frac{x \in Must_S(p, S * x^{\perp}) \qquad p \xrightarrow[S*x^1]{S', k} p'}{\textbf{signal } x \textbf{ in } p \textbf{ end} \xrightarrow[S]{S' \backslash x, k} \textbf{signal } x \textbf{ in } p' \textbf{ end}}$$

$$\text{sig-}\ \frac{x \notin Can^+_S(p, S * x^{\perp}) \qquad p \xrightarrow[S*x^0]{S', k} p'}{\textbf{signal } x \textbf{ in } p \textbf{ end} \xrightarrow[S]{S' \backslash x, k} \textbf{signal } x \textbf{ in } p' \textbf{ end}}$$

$$\text{present+}\ \frac{x^1 \in S \qquad p \xrightarrow[S]{S', k} p'}{\textbf{present } x \textbf{ then } p \textbf{ else } q \textbf{ end} \xrightarrow[S]{S', k} p' \textbf{ end}}$$

$$\text{present-}\ \frac{x^0 \in S \qquad q \xrightarrow[S]{S', k} q'}{\textbf{present } x \textbf{ then } p \textbf{ else } q \textbf{ end} \xrightarrow[S]{S', k} q' \textbf{ end}}$$

Figure 4.1: Semantics rules involving causality analysis in Esterel [Ber02].

to the control flow. For example, the statement `emit x` emits the signal x, in other word it writes *true* inside x. The signal environment computed by $Must_S$ and Can^+_S is a data and code control evaluation: setting a signal to true has a direct impact on the control flow (if it appears in a presence test). In spacetime, the values are not syntactically rooted to the code. Furthermore, the values are not even known by the spacetime compiler since they belong to the host language. Therefore, we perform an analysis over the control flow *only* and without evaluating the value of the variables. This is implemented with the *causal* function that describes the dependencies between the read and write operations in a process. In other word, it attempts to sequentialize a spacetime process by giving a total order between the operations.

In spacetime, each variable can be associated with a counter $(w, rw, r) : LMin \times LMin \times LMin$ indicating that w writes must be done—or invalidated—before rw read-writes and before r reads. In Esterel, this counter becomes $(w, r) : LBool \times LMin$ where a write event is a boolean[3] representing the emission of a signal—emitting several times a signal has no effect. As in spacetime, any number of read can follow. In both, if the counter w does not reach 0 (or *true*), then the reads cannot occur and we have a causality problem. All in all, spacetime is generalizing this analysis to arbitrary lattice values. However, Esterel can perform stronger

[3] We use the lattice *LBool* to define this write event: it evolves from $false$ to $true$.

causality analysis since whenever $w = true$ it also knows that the signal has been emitted, and thus its value.

Valued signals and data variables

In Esterel and ReactiveML, they use a signal environment that contains the set of values emitted on a signal during an instant [Ber00a, Man06]. The values are later aggregated using the combination function of the signal. This composition function is similar to the join in a lattice because it must be associative and commutative. A first minor difference in spacetime is that we aggregate the values immediately and do not keep a set of values. The main difference, is that spacetime allows us to reason about the values with the entailment while in other synchronous languages the manipulation of values is more limited. For example, alternating read and write operations on a variable is usually forbidden. Similarly, Esterel [PBEB07] forbids to write on data variables in distinct processes.

Chapter 5

Behavioral Semantics Across Instants

This chapter formalizes the semantics of the processes progressing across time, and the processes encapsulated in `universe`. A universe lifts the execution of a program in an upper layer of the lattice hierarchy starting from L_4. In essence, a universe is composed of a queue of nodes and of a process that is executed on a faster time scale. For example, admitting that p is a process over L_4 and q a queue of nodes, then `universe with` q `in` p `end` becomes a process over L_5. From the point of view of p, time is flowing faster in L_4 than what another process observes in the upper universe L_5. This idea is not new: time refinement has already been explored in synchronous languages, especially in Quartz [GBS13] and ReactiveML [MPP15]. However, in these languages, each universe is atomically executed, and thus parallel universes do not communicate within their local steps. In spacetime, we execute synchronously parallel universes which can communicate.

The main goal of this chapter is to capture compositionality of universes in semantics rules. In this respect, there are two crucial design aspects:

- **Data compositionality.** Parallel universes can communicate on shared variables defined in a common upper layer. The question is under what spacetime a universe can observe variables defined in its lower and upper universes in order to preserve their monotonic evolutions?
- **Control compositionality.** The simultaneous execution of two universes synchronized on the same time scale with a same queue of nodes.

These two aspects are introduced in the Sections 5.4 and 5.5. To formalize the semantics, we first develop structures extending the behavioral semantics to hierarchical structures (Sections 5.2 and 5.3). The semantics rules developed in Chapter 4 are mostly left unchanged, and can be adapted to the universe extension without complications. Finally, we do not extend the causality analysis to hierarchical structures, and leave this task to future work.

5.1 Composition of parallel universes

We first introduce several design principles for combining universes which are central to the semantics.

Principle 5.1 (Full universe synchronization)**.** *All universes executable in an instant must be executed synchronously, at the same time and at the same rate.*

Corrolary 5.1. *Universes cannot appear in sequence during a same instant.*

Principle 5.2 (Universe composition)**.** *Universes with a same queue of nodes are synchronized in space and time which means that they form only one combined search tree. Universes defined on distinct queues are synchronized in time only, and thus only the completion code is merged in the parallel statements.*

The branches generated by the universes do not conflict: pruning statements are local to the queue of nodes. However, when a universe terminates its execution, the universes defined on others queues might be forced to terminate if they are composed with the disjunctive parallel operator `<>`, otherwise they continue their executions. If one of the universe is pausing up, according to the completion code of parallel operators, every other universe is also pausing up.

Principle 5.3 (Universes communication)**.** *Universes communicate on variables* `single_space` *and any variable defined in a common parent universe. In addition, universes defined on the same queue can communicate through* `world_line` *variables that are transferred in both universes.*

5.2 Universe hierarchy

A challenge with universes is that more than one steps can be executed during a single reaction. Therefore, the variable space S is not sufficient anymore since it describes the value of the variables during a single instant only. To overcome this limitation, we follow the technique used in the semantics of reactive domains in ReactiveML [Pas13, MPP15]. It consists in lifting the space structure to a store $Store(Timestamp, Space)$ where an element $(t, S) \in (Timestamp \times Space)$ is a snapshot of the space S at a unique point in time t. We name this structure the universe hierarchy. To define it, we need intermediate structures in order to precisely define a timestamp.

Definition 5.1 (Timed layer)**.** *A timed layer is a lattice defined by a set of active queues, represented by their names, and an integer indexing the instant of the layer.*

$$TimedLayer(Name) = \langle \mathcal{P}(Name) \times LMax, \quad (Q,i) \models_{TL} (Q',j) \ \mathit{if} \begin{cases} Q \subseteq Q' \\ \wedge \ \ i \models_{LMax} j \\ \wedge \ \ i = j \Rightarrow Q = Q' \end{cases} \rangle$$

where the order $\ell \models \ell'$ is defined if ℓ is happening at the same time or after ℓ', and that ℓ contains less or the same number of active queues than ℓ'. Also, if two timed layers are defined in the same instant, they must be equal (an element in this lattice represents only one layer, not a hierarchy of layers).

Example 5.1 (Timed layer with three universes)**.** Consider the following space-time program:

```
single_space StackLR q;
single_space StackLR r;
par
|| universe with q in pause; end          (A)
|| universe with r in pause; end          (B)
|| universe with r in pause; pause end    (C)
|| universe pause; pause; pause end       (D)
end
```

This program is constituted of two layers, one for the uppermost universe and one that contains four universes named A, B, C and D. In the first instant of the uppermost layer, we have the timed layer $(\{\}, 0)$ with an empty set of queue. The second layer evolves across four instants before all its universes terminate: $(\{q, r\}, 0)$, $(\{q, r\}, 1)$, $(\{r\}, 2)$ and $(\{\}, 3)$. We notice that the last universe D is defined without a queue, which generates an empty set of queue in the fourth instant. The order of a timed layer reflects this temporal evolution, for example we have $(\{r\}, 2) \models (\{q, r\}, 1)$. The number of active queues can only decrease over time. ⌟

A timed layer is only useful for describing the queues in an instant of a single layer. To obtain a precise timestamp of the whole program, we need a set of timed layers, that we call a *timestamp.*

Definition 5.2 (Timestamp)**.** *A timestamp is a store of timed layers such that the set of queues of each layer is distinct (property $(P1)$ below). We guarantee with $(P2)$ that the uppermost universe has an empty set of queues. The index of the store corresponds to the index of the layer in the hierarchy.*

$$\begin{aligned}
&Timestamp(Name) = \langle \\
&\quad \{S \in Store(LMax, TimedLayer(Name)) \mid \\
&\quad\quad (P1) \quad \forall i, j \in \pi_1'(S),\ \pi_1(S(i)) \cap \pi_1(S(j)) = \emptyset \\
&\quad\quad (P2) \quad \pi_1(head(S)) = \{\} \\
&\quad \}, \\
&\quad s ::^{-1} S \models r ::^{-1} R \textit{ if } (s \models_{TL} r) \wedge (s = r \Rightarrow S \models R)
\end{aligned}$$

where we define the operator $::^{-1}$ *as follows:*

$$\begin{aligned}
&::^{-1}\ : TimedLayer \times Timestamp \rightarrow Timestamp \\
&s ::^{-1} S \mapsto alloc^{-1}(S, s)
\end{aligned}$$

where $alloc^{-1}$ *is allocating an element at the minimal location instead of the next maximal location. We also define the operation* $::$

$$:: \; : Timestamp \times TimedLayer \rightarrow Timestamp$$
$$S::s \mapsto alloc(S, s)$$

The two functions $::^{-1}$ and $::$ are used to obtain the first and last timed layers. We give an example to illustrate this timestamp structure.

Example 5.2 (Timestamp of nested universes)**.** Consider the following program:

```
single_space StackLR q;
pause;
universe with q in
  single_space StackLR r;
  pause up;
  universe with r in pause; end
  pause;
end
```

We have a timestamp for each possible point in time across all the layers of this program. We show the set of the generated timestamps ordered by the relation $\models$ in *Timestamp*:

$$t_0 = \{(0, (\{\}, 0))\}$$
$$t_1 = \{(0, (\{\}, 1))\} :: (\{q\}, 0)$$
$$t_2 = \{(0, (\{\}, 1))\}$$
$$t_3 = \{(0, (\{\}, 2))\} :: (\{q\}, 0) :: (\{r\}, 0)$$
$$t_4 = \{(0, (\{\}, 2))\} :: (\{q\}, 0) :: (\{r\}, 1)$$
$$t_5 = \{(0, (\{\}, 2))\} :: (\{q\}, 0)$$
$$t_6 = \{(0, (\{\}, 2))\} :: (\{q\}, 1)$$
$$t_7 = \{(0, (\{\}, 2))\}$$

Importantly, we notice that an instant in an upper layer is ordered "after" the instants in its sub-universe. It defines that an instant only terminates once all of its sub-instants are terminated. ⌟

Now that we can precisely situate a point in time, we describe the *universe hierarchy* that maps timestamp to spaces of variables.

Definition 5.3 (Universe hierarchy)**.** *A universe hierarchy is a store of spaces indexed by timestamps.*

$$UHierarchy(Name, Type) =$$
$$Store(Timestamp(Name), Space(Name, Type))$$

where the order is inherited from Store.

In function of a particular universe hierarchy, we define two notions of "previous timestamp" relatively to a queue and relatively to a layer.

Definition 5.4 (Previous timestamp relative to a queue)**.** *The function $pre^Q(U, t, q)$ maps to the timestamp before t in U relatively to the queue q:*

$$\begin{array}{l} pre^Q : UHierarchy \times Timestamp \times Name \to Timestamp \\ pre^Q(U, t, q) \mapsto \bigsqcup\{t' \in \pi_1'(U) \mid t \models t' \wedge t \neq t' \wedge (t' = t'' :: (Q, i)) \wedge q \in Q\} \end{array}$$

The previous timestamp of a queue is the lowest upper bound of all the timestamp of this queue before t.

Definition 5.5 (Previous timestamp relative to a layer)**.** *The function $pre^L(U, t)$ maps to the previous timestamp in U relatively to the last layer in t:*

$$\begin{array}{l} pre^L : UHierarchy \times Timestamp \to Timestamp \\ pre^L(U, t) \mapsto \bigsqcup\{t' \in \pi_1'(U) \mid t \models t' \wedge t \neq t' \wedge |t| = |t'|\} \end{array}$$

The previous timestamp of a layer is the lowest upper bound of all the timestamp of this layer before t.

Example 5.3 (Previous timestamps)**.** Extending Example 5.2, we have the following time relations with pre^L and pre^Q:

$$\begin{array}{lll} pre^L(U, t_5) = t_1 & pre^Q(U, t_5, q) = t_1 & \\ pre^L(U, t_4) = t_3 & pre^Q(U, t_4, q) = t_1 & pre^Q(U, t_4, r) = t_3 \\ pre^L(U, t_3) = \bot & pre^Q(U, t_3, q) = t_1 & \end{array}$$

⌟

5.3 Hierarchical behavioral semantics

We lift the definitions of branches and completion code to a store indexed by timestamps.

Definition 5.6 (Hierarchical branches)**.** *Given the unordered set of branches B^n, we define the lattice of hierarchical branches as follows:*

$$HBranch(Name) = Store(Timestamp(Name), Store(Name, B^n))$$

where the order is inherited from Store.

The operators $\{\circ, \wedge, \vee\}$ of the algebra B^n are extended to branch hierarchy: they are used to combined the branches with a same universe.

Definition 5.7 (Hierarchical completion code)**.** *Given the completion code integer set $Compl = \{0, 1, 2, 3\}$, its hierarchical lifting is defined as follows:*

$$HCompl(Name) = Store(Timestamp(Name), Compl)$$

where the order is inherited from Store.

We also extend the operators $\{\wedge, \vee\}$ of *Compl* to the completion code hierarchy. The main goal of the completion code hierarchy is to know if one layer paused up or stopped the program. Note that the code is not bound to a particular queue but to a particular layer: it formalizes the idea that universes on different queues are composed in time.

These structures enable us to define the hierarchical version of the behavioral semantics rule as:

$$t, N \vdash p \xrightarrow[U]{U', HB, K} p'$$

where the program p is rewritten into the program p' under the timestamp t, the set of names N and the input/output universe hierarchy U. The effects of this reaction are given in U' which is the output universe hierarchy, HB is the set of branches for each queue and K is the set of completion codes for each layer. We have the relation $U \models U'$ to model that outputs are also inputs of the program.

The rules we introduce in this chapter are heavier than the ones introduced in Chapter 4. This is because they formalize the link between two instants, which is not syntactically represented by spacetime processes. Therefore, to lighten the notation, we introduce several internal statements, so the semantics is better decomposed. We have three internal statements typeset as $\mathsf{transfer}^*$, pop^* and push^*. All these three statements will be elaborated in the course of this chapter.

We can already define the two axioms rules PAUSE-UP and STOP which respectively pause in the upper and outermost universe. This is achieved by setting the completion code to 2 and 3.

PAUSE-UP
$$\texttt{pause up} \xrightarrow[S]{\bot, \langle\rangle, 2} \texttt{nothing}$$

STOP
$$\texttt{stop} \xrightarrow[S]{\bot, \langle\rangle, 3} \texttt{nothing}$$

In the following, we extend the rules of the behavioral semantics only when they significantly differ from the ones defined in Chapter 4. Importantly, we do not formalize the queueing strategy where the queue is a `world_line` variable, and we leave it to future work. However, there exists very few search strategy that need this capability, and none of our strategies in Chapter 6 needs `world_line` queues.

5.4 Hierarchical variable

In Chapter 4, we have a unique space of variables in an instant, and thus a variable only exists in one layer at a time. Extending the semantics to universes also induces that a variable can exist in several layers of the universe hierarchy. There are two reasons we want to define a variable in several layers:

- Communication between sub-universes through variables defined in a common upper universe.

- Access to results obtained in a sub-universe.

However, we cannot arbitrarily transfer one variable across layers: we must ensure that the monotonicity of the computation is preserved. The Section 5.4.2 tackles the case where a variable is declared in a universe and transferred into one of its sub-universes. The opposite case where we access the value of a variable of a sub-universe is introduced in Section 5.4.3. Before these two, it is necessary to define the read and write operations on variables defined in multiple layers of a hierarchy.

5.4.1 Read, write and allocate variables in the hierarchy

We read a variable x at a particular point in time t, and we join the values of all the variables defined in the upper layers:

$$\begin{aligned}&\mathit{read} : \mathit{UHierarchy} \times \mathit{Timestamp} \times \mathit{Name} \rightarrow \mathit{Type}\\ &\mathit{read}(U,t,x) \mapsto \bigsqcup\{U(t')^V(x) \mid t' \subseteq t\}\end{aligned}$$

We refine the rule VAR as follows:

$$\textsc{var}\quad t \vdash x \overset{\bot}{\underset{U}{\hookrightarrow}} \mathit{read}(U,t,x)$$

An advantage of the universes is that we can define the operator `pre` x that retrieves the value of a variable at its former instant.

$$\begin{aligned}&\mathit{pre}^V : \mathit{UHierarchy} \times \mathit{Timestamp} \times \mathit{Name} \rightarrow \mathit{Type}\\ &\mathit{pre}^V(U,t,x) \mapsto \begin{cases}\bot & \text{if } \mathit{pre}^L(U,t) = \bot\\ \mathit{pre}^V(U,t',x) & \text{if } (x,a) \notin U(t')^V \text{ where } t' = \mathit{pre}^L(U,t)\\ \mathit{read}(U,\mathit{pre}^L(U,t),x) & \text{otherwise}\end{cases}\end{aligned}$$

The function pre^V maps to the value of a variable at the former instant where it was defined. The rule is then defined as follows:

$$\textsc{pre}\quad \frac{U(t)^{st}(x) =\!\!\rightarrow}{t \vdash \texttt{pre}\ x \overset{\bot}{\underset{U}{\hookrightarrow}} \mathit{pre}^V(U,t,x)}$$

We define the operator `pre` only for `single_space` variable, and we leave the case of `world_line` for future work.

We write into a variable only in its current defining layer, unless it is a `single_space` variable, in which case we propagate the change through the upper layers:

$$\begin{aligned}
&\mathit{write} : \mathit{UHierarchy} \times \mathit{Timestamp} \times \mathit{Name} \times \mathit{Type} \to \mathit{UHierarchy} \\
&\mathit{write}(U, t, x, v) \mapsto \\
&\begin{cases}
\{(t, \{(x, (v, \to))\})\} \sqcup \mathit{write}(U, t', x, v) \text{ where } t = t' :: \ell, & \text{if } (x, (\to, v')) \in U(t) \\
\{(t, \{(x, (v, st))\})\} & \text{if } (x, (st, v')) \in U(t) \\
& \wedge\, st \neq \to \\
\bot & \text{otherwise}
\end{cases}
\end{aligned}$$

The write operation refines the semantics rule of the tell statement:

$$\textsc{tell}\quad \frac{t \vdash e \overset{U'}{\underset{U}{\hookrightarrow}} v \qquad U'' = \mathit{write}(U, t, x, v)}{t, \{\} \vdash x \texttt{ <- } e \xrightarrow[U]{U' \sqcup U'',\, \{\},\, \{(t,0)\}} \texttt{nothing}}$$

It is similar to the rule TELL in Chapter 4, but we rely on the function *write* instead of writing directly into the space.

Taking into account the hierarchical view, we also need to redefine how a variable is declared. Our approach is to declare a variable in the universe hierarchy with the current timestamp. We initialize the variable to a different value depending on its spacetime:

- In the case of `single_time` variables, they have a different name in every instant and so their values is equal to $\bot$.
- The `world_line` variables are handled in the rule POP where we set their values according to the node popped from the queue.
- As for `single_space` variables, we retrieve their value from the latest instant.

This behavior is specified in the following function *allocate*:

$$\begin{aligned}
&\mathit{allocate} : \mathit{UHierarchy} \times \mathit{Timestamp} \times \mathit{Spacetime} \times \mathit{Name} \to \mathit{UHierarchy} \\
&\mathit{allocate}(U, t, x, st) \mapsto \begin{cases}
\{(t, \{(x, (\bot, st))\})\} & \text{if } st \neq \to \\
\{(t, \{(x, (\mathit{pre}^V(U, t, x), \to))\}) & \text{if } st = \to
\end{cases}
\end{aligned}$$

We extend the rule VAR-DECL with this new function:

$$\textsc{var-decl}\quad \frac{t, N \vdash p[x \to n] \xrightarrow[U]{U',\, HB,\, K} p' \qquad U'' = \mathit{allocate}(U, t, n, st)}{t, N \mathbin{\dot\cup} \{n\} \vdash st\ \mathit{Type}\ x\ ;\ p \xrightarrow[U]{U' \sqcup U'',\, HB,\, K} st\ \mathit{Type}\ x\ ;\ p'}$$

5.4.2 Top-down spacetime transfer

A variable can be transferred from one universe U_i to a lower universe U_{i-1}, and the variable can be seen with different spacetime in both universes. We define a top-down transfer as a tuple $(source, queue, target)$ where *source* is the spacetime of the variable in U_i, *queue* is the spacetime of the queue of the universe being traversed, and *target* is the spacetime of the variable in U_{i-1}. For example, we write $(\downarrow, \circlearrowleft, \rightarrow)$ for a `world_line` variable in U_i traversing a universe with a `single_time` queue, and being transferred as a `single_space` variable in U_{i-1}.

We have $3 \times 3 \times 3$ possible combinations of possible top-down transfers. Due to this relatively high number of transfers, it is worth to have a precise semantics of data transfers between universes. The main goal of this semantics is to preserve the monotonic evolution of the computation and variables through multiple points of view. As shown in Chapter 4, non-monotonicity can lead to non-causal programs that are non-reactive and generate indeterminism. Therefore, we formalize variable transfer in order to avoid these issues.

In order to reduce the number of transfers to consider, we first study all the combinations involving an occurrence of a `single_time` variable. We make two observations:

- The target cannot be a `single_time` variable because such a variable only exists in a single instant, but it would exist in several instants of the parent's universe.
- If the source is a `single_time` variable, then the queue must be declared as `single_time` as well. It prevents the `single_time` variable inside the sub-universe to live longer than the parent's one.

Therefore, the only valid combinations where a `single_time` spacetime occurs are:

$$\left(\left(\begin{smallmatrix}\circlearrowleft\\ \downarrow\\ \rightarrow\end{smallmatrix}\right), \circlearrowleft, \left(\begin{smallmatrix}\downarrow\\ \rightarrow\end{smallmatrix}\right)\right)$$

which is natural since a `single_time` queue indicates that the transferred variables only live for one instant, and thus cannot exceed the lifetime of the parent's variable. Also, we notice that a `world_line` variable can be a target, and thus the question: how do we merge the variable back in the parent universe? We do not merge it back. As shown in the former section, a `world_line` variable can reside in several queues belonging to particular layers.

Without the `single_time` variables, we are left to $2 \times 2 \times 2$ possibilities among which 6 are valid. We first explain the 2 possibilities that are not valid:

$$\left(\downarrow, \rightarrow, \left(\begin{smallmatrix}\downarrow\\ \rightarrow\end{smallmatrix}\right)\right)$$

A universe with a `single_space` queue of nodes contains statements such as `pause up` which make the universe live for more than one instant in its upper universe. A consequence is that the variable defined in U_i must live for at least as long as

the variable defined in U_{i-1}. This is not the case when the source variable has a `world_line` spacetime. In the case of a `world_line` target, the problem occurs in the tree of the sub-universe: the variable does not evolve monotonically along a path since we backtrack in the parent universe without backtracking in the sub-tree. Hence, two nodes on a same path do not necessarily have the same root node, and thus the same amount of information. In the case of a single space target, it is supposed to be global to the queue but is backtracked nevertheless in the parent universe. Hence, the variable does not evolve monotonically according to its single space spacetime. We now consider the remaining 6 cases corresponding each to a specific situation.

The two first represent a variable that evolves globally to the search trees of U_i and U_{i-1}. In this case, the spacetime of the queue does not impact the meaning of the transfer.

$$(\rightarrow, \left(\substack{\downarrow \\ \rightarrow}\right), \rightarrow)$$

The third transfer mode is a `world_line` variable evolving locally to a path in the search tree of U_{i-1} but globally to the search tree of U_i. It can be useful to keep a store of global information in U_i. The monotonicity of the computation is preserved because the local values in the variables of U_{i-1} are not observed by U_i—remember that the `world_line` is associated to its queue and thus copied into U_{i-1}.

$$(\rightarrow, \rightarrow, \downarrow)$$

The next transfer is similar to the third with the difference that the queue is backtracked in U_i. It is still monotonic because the value of the root node—given by the source single space variable—evolves monotonically.

$$(\rightarrow, \downarrow, \downarrow)$$

The fifth transfer captures a variable that is global to the search tree in U_{i-1} while being backtracked in U_i. In other terms, the variable is backtracked whenever the queue is backtracked but not on a backtrack in the search tree of U_{i-1}. It models a property global to U_{i-1} along a path of the search tree of U_i.

$$(\downarrow, \downarrow, \rightarrow)$$

The sixth and last transfer indicates that a variable is backtracked whenever a backtrack occurs in U_i or U_{i-1}. It models a property local to a path in the search tree of U_{i-1} embedded in the one of U_i.

$$(\downarrow, \downarrow, \downarrow)$$

Finally, we define a rule that abstracts over the valid top-down transfers:

$$\frac{\substack{\textsc{TopDownTransfer} \\ (U(t)^{st}(x), U(t)^{st}(q), target) \in \left\{ \begin{array}{l} ((\substack{\circlearrowleft \\ \downarrow \\ \rightarrow}), \circlearrowleft, (\substack{\downarrow \\ \rightarrow})) \\ ((\substack{\downarrow \\ \rightarrow}), \downarrow, (\substack{\downarrow \\ \rightarrow})) \\ (\rightarrow, \rightarrow, (\substack{\downarrow \\ \rightarrow})) \end{array} \right. \\ U' = allocate(U, t, target, x)}}{t, \{\} \vdash \mathsf{transfer}^*(q, target, x) \xrightarrow[U]{U', \{\}, \{(t,0)\}} \mathtt{nothing}}$$

We rely on the internal statement $\mathsf{transfer}^*$ to organize this verification as a semantics rule.

5.4.3 Bottom-up spacetime transfer

We introduce the notion of bottom-up transfer which enables a universe to retrieve data computed in their sub-universes. This is useful for retrieving the values of `world_line` variables which are not observable in the parent universe. We can access to the value of a `world_line` variable x in a lower layer defined by a queue q with the syntax $q::x$. It is formalized by the following rule:

$$\frac{\substack{\textsc{BottomUpTransfer} \\ v = U(pre^Q(U, t, q))^V(x)}}{t, \{\} \vdash q::x \overset{\perp}{\underset{U}{\hookrightarrow}} v}$$

where we retrieve the latest timestamp relatively to the queue q, so we have the value of x in the latest instant of its universe.

5.5 Semantics of the rules across time

We formalize the semantics rules that make the execution of a program progresses across instants.

5.5.1 Semantics of the queueing strategy

We bridge the space of two successive instants with the queueing strategy. It pushes and pops nodes into and from the relevant queues. We first define the structure of a *future* and then the rules to push and pop nodes from a queue. A future encapsulates a branch process and its spaces.

Definition 5.8 (Future)**.** *A future is a tuple* $(S^{\downarrow}, S^{\circlearrowleft}, b)$ *where* $S^{\downarrow}$ *is a set of* `world_line` *variables,* $S^{\circlearrowleft}$ *is a set of* `single_time` *variables and* b *is a branch process.*

We propose a semantics rule Pop to extract a future from the queue and instantiate it. This rule is summoned inside the universe rule that we introduce later. We tackle the first instant of a universe by checking if the queue was defined in a previous instant (rule Pop) or not (rule PopFirst).

$$\textsc{PopFirst}\quad \frac{pre^Q(U,t,q) = \bot \qquad t, N \vdash p \xrightarrow[U]{U',HB,K} p'}{t, N \vdash \mathsf{pop}^*\ p \xrightarrow[U]{U',HB,K} p'}$$

$$\textsc{Pop}\quad \frac{\begin{array}{c} t, N \vdash b \xrightarrow[U]{U'',\{\},\{(t,0)\}} b' \qquad t, N \vdash p \xrightarrow[U]{U''',HB,K} p' \\ (U'(t)^{\downarrow}, U'(t)^{\circlearrowleft}, b) = \pi_2(pop(U(t')(q))) \qquad t' = pre^Q(U,t,q) \qquad t = t'' :: (\{q\}, i) \end{array}}{t, N \vdash \mathsf{pop}^*\ p \xrightarrow[U]{U' \sqcup U'' \sqcup U''',HB,K} p'}$$

The queue q is the queue that is at the tail of the current timestamp: the one under which the universe is currently executed. We use the function pre^Q to retrieve the value of the queue q at its previous instant, and we extract a node from this queue.

An important invariant of the semantics rules is that the queues in the timestamp t are always singleton sets. This is because a semantics rule is always executed under a single universe per layer. Therefore, we cannot push futures locally onto the queue inside a universe since these futures might be composed with futures created by neighbour universes defined on the same queue. Our solution is to push the futures onto the queue at the root of the semantics rule, after they have been composed. This is also the reason we need the hierarchical branches structure, in order to aggregate all the branches created by all the universes. We first define a function that creates a set of futures and pushes them on the relevant queue:

$$\begin{array}{l} future(U,t,q,B) = push(Q, \{(j, (U(t)^{\downarrow}, U(t)^{\circlearrowleft}, b)) \mid (\mathsf{space}\ \ b)_j \in B\}) \\ \qquad \text{where } Q = \pi_1(pop(U(pre^Q(U,t,q))(q))) \text{ such that } t = t' :: (\{q\}, i) \end{array}$$

The link between two successive instants is done in two steps:

(i) We pop a node from the queue q at the previous timestamp, and we keep the queue without this node—instead of retrieving the value extracted as in Pop.

(ii) We push the created futures onto the queue retrieved at step (i).

The semantics rule Push first derives the process p, and from the hierarchical branches created by p, we create a set of futures for each timestamp of every queue appearing in HB.

Push

$$\frac{t, N \vdash p \xrightarrow[U]{U', HB, K} p' \qquad U'' = \bigsqcup\{\forall(t', bs) \in HB,\ \forall(q, B) \in bs,\ \{(t', future(U, t', q, B))\}\}}{t, N \vdash \mathsf{push}^*\ p \xrightarrow[U]{U' \sqcup U'', HB, K} p'}$$

5.5.2 Universe statement

The well-formedness of universes is checked by the rule Universe. This rule represents a universe that is executed in the current instant of the parent universe, we tackle the local step of a universe in another rule explained below.

Universe

$$\frac{\begin{array}{c} t :: (\{q\}, 0), N \vdash p \overset{U', HB, K}{\underset{U}{\Longrightarrow}} p' \\ \forall(st_i, x_i) \in \vec{X},\ t, \{\} \vdash \mathsf{transfer}^*(q, st_i, x_i) \xrightarrow[U]{U''_i, \{\}, \{(t,0)\}} \texttt{nothing} \\ U'' = U' \sqcup \bigsqcup_{i \leq |X|} U''_i \qquad K' = K \sqcup \{(t, k)\} \\ k = K(pre^Q(U, t, q)) \qquad \text{if } U(t)^{st}(q) = \circlearrowleft \text{ then } k = 0 \text{ must hold} \end{array}}{t, N \vdash \texttt{universe(}\vec{X}\texttt{) with } q \texttt{ in } p \texttt{ end} \xrightarrow[U]{U'', HB, K'} \texttt{universe(}\vec{X}\texttt{) with } q \texttt{ in } p' \texttt{ end}}$$

This rule encapsulates the execution a process with a dedicated queue and starts its execution from the instant 0 of the next layer of the current timestamp. The completion code of the universe is the same as the one of the latest local step relatively to the queue q. We verify that every universe with a `single_time` queue immediately terminates. Finally, all the variable top down transfers are verified, and the universe hierarchies generated by $transfer$ are aggregated.

The transition $\Rightarrow$ is used to describe the local steps describing the execution of a universe. To define $\Rightarrow$, we use the three rules GlobalStep, QueueEmpty and EndOfInstant. The rule GlobalStep executes multiple steps of a statement and increases its instant counter between each step.

GlobalStep

$$\frac{\begin{array}{c} t, N \vdash \mathsf{pop}^*\ p \xrightarrow[U]{U', HB, K} p' \qquad t' :: (Q, i+1), N \vdash p' \overset{U'', HB', K'}{\underset{U}{\Longrightarrow}} p'' \\ K(t) = 1 \qquad t = t' :: (Q, i) \qquad U(t)^V(q) \neq \bot \end{array}}{t, N \vdash p \overset{U' \sqcup U'', HB \sqcup HB', K \sqcup K'}{\underset{U}{\Longrightarrow}} p''}$$

We can only apply this rule if the current process pauses into the current instant ($k = 1$), and the queue is not empty. Finally, the transition for the local step is evaluated with a timestamp t to identify the current queue and instant.

We detect the end of an instant in the rule EndOfInstant whenever the completion code of a step is different from 1.

$$\textsc{EndOfInstant}\quad \frac{t, N \vdash \mathsf{pop}^*\ p \xrightarrow[U]{U', HB, K} p' \qquad K(t) \neq 1}{t, N \vdash p \overset{U', HB, K}{\underset{U}{\Longrightarrow}} p'}$$

Alternatively, in rule EmptyQueue, we detect the termination of the process whenever its queue of nodes is empty. In this case, we set the completion code of the transition to 0.

$$\textsc{EmptyQueue}\quad \frac{t, N \vdash \mathsf{pop}^*\ p \xrightarrow[U]{U', HB, K} p' \qquad U(t)^V(q) = \bot}{t, N \vdash p \overset{U', HB, K \sqcup \{(t,0)\}}{\underset{U}{\Longrightarrow}} p'}$$

5.5.3 Reaction rule

The reaction rule executes every local step until we pause up the computation in the top-level universe, and hand the control over to the user. The reaction transition $\hookrightarrow$ is the uppermost transition of the program and it is responsible for verifying the well-formedness of the queues of every layer.

$$\textsc{React}\quad \frac{\begin{array}{c} t, N \vdash \mathsf{push}^*\ p \xrightarrow[U]{U', HB, K} p' \\ \{(0, (\{\}, i+1))\}, N \vdash p' \overset{U'', k}{\underset{U}{\hookrightarrow}} p'' \qquad K(t) = 1 \qquad t = \{(0, (\{\}, i))\} \end{array}}{t, N \vdash p \overset{U' \sqcup U'' \sqcup U''', 1}{\underset{U}{\hookrightarrow}} p''}$$

$$\textsc{ReactEnd}\quad \frac{t, N \vdash \mathsf{push}^*\ p \xrightarrow[U]{U', HB, K} p' \qquad k = K(t) \qquad k \neq 1}{t, N \vdash p \overset{U' \sqcup U'', k}{\underset{U}{\hookrightarrow}} p'}$$

As mentioned before, the upper layer is the user environment, and therefore it contains an empty set of queue since it cannot be backtracked. The rule React automatically executes the next instant of the program p if it is paused. We stop reacting whenever the program is paused up, stopped or terminated in the (current) uppermost universe (rule ReactEnd). Note that the very first reaction is initialized with the bottom timestamp, which is equal to $\{(0, (\{\}, 0))\}$.

Part III

Applications

Chapter 6

Modular Search Strategies

6.1 Pruning strategies

In this section, we first consider processes maintaining statistics of the search, and then use them to prune the search tree at some points.

6.1.1 Statistics

A search language usually relies on some statistics given by the constraint solver. We show in this section that these statistics can be programmed directly in spacetime. There are two advantages:

(i) To stay independent from the constraint solver which is used for propagation only.

(ii) To be able to use the statistics directly as a communication channel between two processes.

We isolate four counters in distinct classes for counting the total number of nodes, the depth, the discrepancies and the number of backtracks. The associated counters when exploring the tree from left to right in depth-first search are depicted in Figure 6.1.

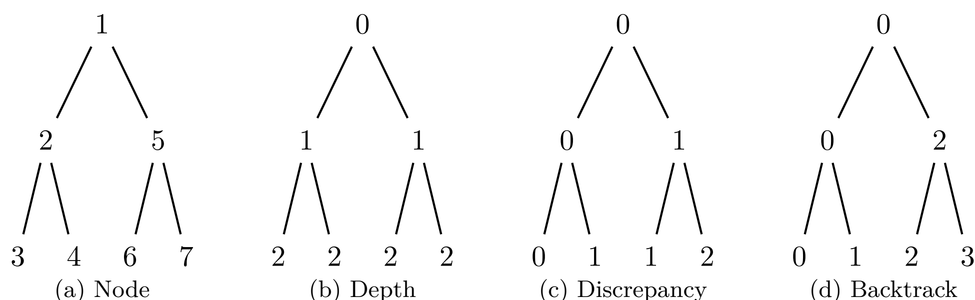


Figure 6.1: The evolution of exploration statistics during a depth-first search.

```
class Node {
  single_space LMax value = new LMax(0);
  public flow count = readwrite value.inc();
}
class Depth {
  world_line LMax value = new LMax(0);
  public proc count =
    pause;
    flow readwrite value.inc() end
}
class Discrepancy {
  world_line LMax value = new LMax(0);
  public flow count =
    space nothing end;
    space readwrite value.inc() end
}
class Backtrack {
  single_space LMax value = new LMax(0);
  public flow count =
    space nothing end;
    space readwrite value.inc() end
}
```

Every counter maintains a variable `value` of type `LMax`—the lattice of increasing integers. The lattice `LMax` exposes a monotonic function $inc(v) := v \sqcup (v + 1)$ incrementing by one the current value of the counter. A first observation is that modifying the spacetime specifier of `value` impacts on the statistics computed. For example, `Node` uses a `single_space` counter that is incremented in every node, while `Depth` uses a `world_line` counter incremented along a path and restored upon backtracking. The only other difference is that we keep a depth of 0 during the first instant, whereas the node's counter is incremented. The next two counters, computing the numbers of discrepancies and backtracks, evolve according to the branch taken. Assuming that the tree is explored from left to right, we do not increment the counter in the first branch, which is modelled with `space nothing end`. These counters are incremented each time we take a right branch with the difference that the discrepancies' counter evolves along a path and the backtracks' counter evolves along the whole search tree.

These statistics are virtually computed in every solver. They serve to stop the search early in bounded search strategies and to obtain information about why a search strategy is slow—for example if the number of backtracks is abnormally high.

6.1.2 Bounded search

When exploring large search trees, it is often required to bound the search to some limits in order to avoid waiting too long for a solution. In particular, with optimization problems, the best solution "obtained so far" can be returned if we

need a solution quickly instead of the best one. Therefore, we can use the statistics to prune the search tree after reaching various limit. For example, the following process enforces a limit on the number of nodes:

```
class BoundedNode {
  single_space LMax limit;
  single_space Node nodes = new Node();
  public BoundedNode(LMax limit) { ... }

  public flow bound =
    when nodes.value |= limit then
      prune
    end
}
```

We initialize the class `BoundedNode` with a limit and we prune the search tree if we reach this limit. The statement `prune` has the effect to repeatedly discard every node from the queue. Importantly, this strategy is kept independent from the actual problem being solved, and thus it achieves modularity since it can be reused with any other strategy.

Similarly, using the depth statistics and the statement `prune`, we can specify a search tree pruned at a certain depth:

```
class BoundedDepth {
  single_space LMax limit;
  single_space Depth depth = new Depth();
  public BoundedDepth(LMax limit) { ... }

  public flow bound =
    when depth.value |= limit then
      prune
    end
}
```

A last example is to bound the search tree by its number of discrepancies:

```
class BoundedDiscrepancy {
  single_space LMax limit;
  single_space Discrepancy discrepancies = new Discrepancy();
  public BoundedDiscrepancy(LMax limit) { ... }

  public flow bound =
    space nothing end;
    when discrepancies.value |= limit then prune end
}
```

This process is designed such that we only discard the right branch if taking this branch would exceed the discrepancy's limit. When the limit condition is not entailed, the process returns a single branch labelled with the process `nothing` that has a neutral effect. Therefore, we do not push the left branch onto the

queue if its number of allowed discrepancies is exceeded. Pushing branches that are immediately discarded was a problem encountered with the extension `tor` in Section 2.6, and described by the second issue in the same section.

6.2 Restart-based search strategies

A restart-based search strategy is any strategy that, at some points, restarts the search at the root node. In the lattice hierarchy, such strategies are captured in the lattice L_5. We build two restart-based search strategies that incrementally explore the search tree, and we illustrate a strategy in L_6 that combines these two strategies.

6.2.1 L_5 search strategies

Restart-based search strategies restart the whole search when reaching a limit, and thus produce a sequence of search trees. Such strategies are defined over the lattice L_5 in the hierarchy introduced in Section 1.3.

Iterative deepening search

Iterative deepening search (IDS) [Kor85] restarts the search after reaching a depth bound. We increase the depth on each restart to explore the tree more deeply. We show the trees obtained after three iterations in Figure 6.2, the dashed line are the part of the tree unexplored (and pruned at this iteration). The class `IDS` implements this strategy:

```
class IDS {
  single_space LMax limit = new LMax(0);

  public proc search(q) =
    loop
      universe(single_space limit) with q in
        single_space BoundedDepth depth = new BoundedDepth(limit);
        depth.bound()
      end
      pause;
      readwrite limit.inc();
    end
}
```

The exploration of a search tree is encapsulated inside a universe: it executes the depth-bounded search until the queue of this universe is empty. When the search is over, the limit is incremented and the search restarts with this new limit. Importantly, we do not specify when to stop the restart iterations. The strategy is thus generic to any kind of problems—including non-CSP's ones. In the next section, we isolate a termination condition, based on the exhaustiveness of the search, in a distinct process.

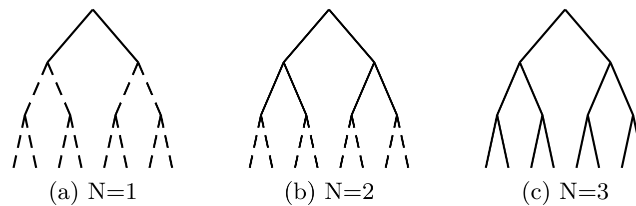


Figure 6.2: Trace of the iterations of IDS.

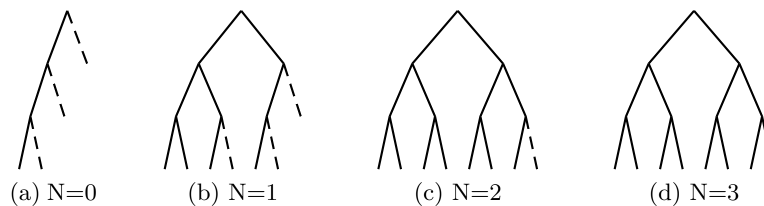


Figure 6.3: Trace of the iterations of LDS.

Limited discrepancy search

Limited discrepancy search (LDS) [HG95] works similarly to IDS with a discrepancies bound instead of a depth bound. This strategy assumes that the left branch is the preferred one—according to some heuristics—and it limits the number of times the search can deviate from this preferred choice. It is illustrated in the Figure 6.3 and given by the following spacetime program:

```
class LDS {
  single_space LMax limit = new LMax(0);

  public flow search(q) =
    readwrite limit.inc();
    universe(single_space limit) with q in
      single_space BoundedDiscrepancy discrepancies = new BoundedDiscrepancy(limit);
      discrepancies.bound()
    end
}
```

We rely on `BoundedDiscrepancy` to prune the part of the search tree that exceeds the limit of discrepancies.

6.2.2 Exhaustiveness and state space decomposition

As mentioned previously, the restart strategies `IDS` and `LDS` are defined generically for any problem. Therefore, they do not terminate since the stop criterion is generally proper to a problem. For CSP, we introduced the state space decomposition in the Section 1.3 which separates the search space into the unknown, failed and

solution spaces. It gives us the ground to design a stop criterion based on exhaustiveness: the search terminates whenever we entirely explored the unknown space. First, we introduce two new statistics counters to compute the cardinality of the state space decomposition of L_4:

```
class SolutionNode {
  public single_space LMax value = new LMax(0);
  world_line VStore domains;
  world_line CStore constraints;

  public SolutionNode(VStore domains, CStore constraints) { ... }

  public flow count =
    single_time ES consistent = read constraints.consistent(read domains);
    when consistent == true then
      readwrite value.inc()
    end
}
```

The class `SolutionNode` reflects the number of explored nodes that belong to the solution space. Similarly, we propose the class `FailNode` for computing the cardinality of the explored failed space:

```
class FailNode {
  public single_space LMax value = new LMax(0);
  world_line VStore domains;
  world_line CStore constraints;

  public FailNode(VStore domains, CStore constraints) { ... }

  public flow count =
    single_time ES consistent = read constraints.consistent(read domains);
    when consistent == false then
      readwrite value.inc()
    end
}
```

Therefore, the total number of nodes minus the solution and fail counters gives the cardinality of the currently explored unknown space. The problem is that the nodes intern to the search tree will be counted in, but we are only interested by the nodes currently inside the queue. To obtain the number of unexpanded nodes, we first need to count the number of internal nodes. This is given by the following class:

```
class InternalNode {
  public single_space LMax value = new LMax(0);
  public flow count =
    single_space LBool branch_taken = false;
    space
      when branch_taken == false then
```

```
        readwrite value.inc();
        branch_taken <- true;
      end
    end
}
```

The node that is instantiated in the current instant is not an internal node yet. Hence, we increment the counter `value` as soon as we instantiate a child of this node. In case the current node has several children, we increase the internal node counter only once, and thus we use the variable `branch_taken` to indicate when a children has been instantiated.

The number of unexpanded nodes is the difference between the unknown state space and the internal nodes. These counters enable us to compute a boolean flag indicating if the search is exhaustive or not:

```
class Exhaustiveness {
  public single_space LBool value = false;
  world_line VStore domains;
  world_line CStore constraints;
  single_space InternalNode int_nodes = new InternalNode();
  single_space Node nodes = new Node();
  single_space SolutionNode sols;
  single_space FailNode fails;

  public Exhaustiveness(VStore domains, CStore constraints) { ... }

  public proc detect =
    par
    || int_nodes.count()
    || nodes.count()
    || sols.count()
    || fails.count()
    || exhaustive()
    end

  flow exhaustive =
    single_time LMin unexpanded =
      count_unexpanded(read nodes, read int_nodes, read sols, read fails);
    when unexpanded |= 0 then
      value <- true
    end

  LMin count_unexpanded(LMax nodes, LMax int_nodes, LMax sols, LMax fails) {
    return new LMin(nodes.get() - int_nodes.get() - sols.get() - fails.get());
  }
}
```

We rely on the Java method `count_unexpanded` to perform the subtraction and returns the number of unexpanded nodes. We set the boolean `exhaustive` to *true*

whenever this counter reaches 0. In this case, we are sure that every node has been expanded since the frontier of the tree only contains solution and failed nodes.

6.2.3 Composing L_5 search strategies

Spacetime offers a well-defined semantics for composing search strategies in L_5 with high-level operators. To demonstrate this, we combine in different ways the search strategies developed in this section. First of all, we start with a search strategy IDS stopping when we reach the first solution or exhaustively explored the search space:

```
class IDS_CSP {
  single_space VStore domains;
  single_space CStore constraints;
  single_space LBool exhaustive = new LBool(false);
  single_time StackLR queue = new StackLR();

  public IDS_CSP(VStore domains, CStore constraints) { ... }

  public proc search =
    weak abort when exhaustive |= true in
      par
      <> base_search()
      <> exhaustiveness()
      <> restart()
      end
    end

  proc restart =
    single_space IDS ids = new IDS();
    ids.search(queue)

  flow base_search =
    universe(world_line domains, world_line constraints) with queue in
      single_space Solver solver = new Solver(domains, constraints);
      solver.first_solution();
    end

  flow exhaustiveness =
    universe(world_line domains, world_line constraints,
             single_space exhaustive) with queue in
      single_space Exhaustiveness e = new Exhaustiveness(domains, constraints);
      par
      || e.detect()
      || exhaustive <- e.value;
      end
    end
}
```

There are several key points in this example:

- In the process `search`, we abort the computation whenever the flag `exhaustive` reaches *true*. It is a weak abort for the same reason as in `Solver`.
- The processes are composed using the conjunctive parallel operator, because whenever `base_search` reaches a solution, we want to stop the search. Moreover, this conjunctive composition is propagated inside the universe: we compose the branches created by `Solver` and the pruning strategy of `IDS` by intersection.
- The queue of nodes `queue` has the spacetime specifier `single_time`, and thus it is reinitialized at each iteration of IDS.
- The computation of the base search and exhaustiveness are encapsulated into two universes which are automatically synchronized with the one of the strategy IDS.
- The variable `domains` and `constraints` are modified locally in the universe, and thus the ones declared as attribute of `IDS_CSP` are not modified. Therefore, each iteration starts with the initial CSP.

The strategy `IDS_CSP` combines trees produced by the two search strategies `Solver` and `IDS`, and thus we combine strategies defined over L_4. We extend this search strategy by combining the two restart-based search strategies IDS and LDS. It lifts the composition to sequences of trees, and thus we combine strategies defined over L_5. For example, in Figures 6.4 and 6.5, we compose IDS and LDS by the intersection and union of their search trees. This is obtained by the following spacetime program (extending `IDS_CSP`):

```
class IDS_LDS {
  single_time StackLR queue = new StackLR();

  // same as IDS_CSP...

  // By default, we perform the intersection.
  proc restart =
    single_space IDS ids = new IDS();
    single_space LDS lds = new LDS();
    restart_intersect(ids, lds)

  proc intersect(ids, lds) = par ids.search(queue) <> lds.search(queue) end
  proc union(ids, lds) = par ids.search(queue) || lds.search(queue) end
}
```

The conjunctive and disjunctive parallel operators compute respectively the intersection and the union of the search trees produced by IDS and LDS.

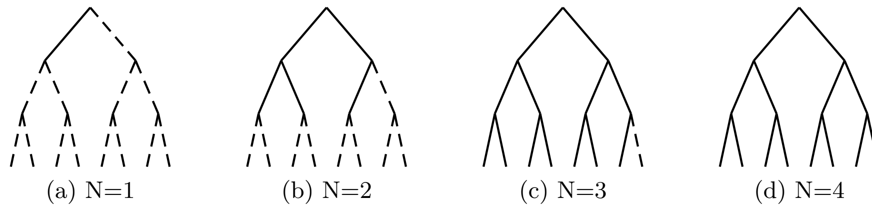


Figure 6.4: Intersection of IDS and LDS.

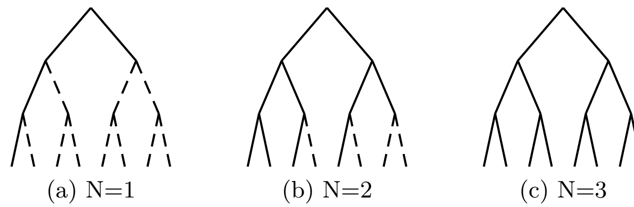


Figure 6.5: Union of IDS and LDS.

6.2.4 L_6 search strategy

We demonstrate a search strategy over the lattice L_6. We launch a LDS strategy until a restart threshold is reached, and then we continue with the IDS strategy if LDS was not exhaustive.

```
class LDS_fby_IDS {
  single_space VStore domains;
  single_space CStore constraints;
  single_space LMax restarts_limit;
  single_space IDS_CSP ids_csp;

  public LDS_fby_IDS(VStore domains, CStore constraints, LMax restarts_limit) { ... }

  proc search =
    single_time ES continue;
    universe(single_space domains, single_space constraints,
             single_space restarts_limit, single_space continue) in
      single_space BoundedNode restarts = new BoundedNode(restarts_limit);
      single_space LDS_CSP lds_csp = new LDS_CSP(domains, constraints);
      par
      <> restarts.bound()
      <> lds_csp.search()
      end;
      continue <- (read lds_csp.queue::constraints).consistent(read lds_csp.queue::domains)
        and lds_csp.exhaustive.value == false;
    end;
    when continue then
      ids_csp.search()
```

```
    end
}
```

We reuse the class BoundedNode to prune the search after a number of restarts. The conjunctive parallel will terminate when we reach the restart bound. Moreover, we use the conjunctive parallel to exit the universe if the search is exhaustive with the LDS strategy. Once we exit the LDS search, we test that we did not reach a solution and that the search was not exhaustive, in this case we continue with IDS.

6.3 Guarded commands

We give an example of the Dijkstra's guarded commands [Dij75] in spacetime and show that we can also program "don't-care nondeterministic" computations.

```
proc guarded_spaces =
  when c1 then space p1 end ;
  when c2 then space p2 end ;
  when c3 then space p3 end
```

For instance, assuming the conditions c_1 and c_3 are true, the sequence of branches $\langle p_1, p_3 \rangle$ is created. Using this basic process, we can simulate "don't-care nondeterminism". This can be achieved with a particular queueing strategy discarding all nodes but one, or directly in the language as follows:

```
single_time LMax alt = new LMax(0);

proc do_not_care_nondeterminism =
  par
  || guarded_spaces()
  || count_alternatives()
  || select_one()
  end

proc count_alternatives =
  when c1 then alt.inc() end ;
  when c2 then alt.inc() end ;
  when c3 then alt.inc() end

proc select_one =
  single_time LMax choice = bot;
  choice <- select(0, alt);
  for(;)(single_time LMax current : new Range(1, alt))
    when current == choice then
      space nothing end
    else prune end
  end
```

The variable `alt` is a counter of the entailed commands. The process `count_alternatives`, using the same conditions as in the process `guarded_spaces`, increments the variable `alt` for each entailed alternative. From these alternatives, one is selected according to a host function `select` in the process `select_one`, and we store its index in the variable `choice`. Using the loop `for`, we provide a "mask sequence" such that every branch is pruned but the one selected—this is represented by the statement `space nothing end`. Hence, only one branch is actually added onto the queue. For example, if c_1 and c_3 are entailed and c_3 is chosen, the sequence given by the process `guarded_spaces` is $\langle p_1, p_3 \rangle$ and the one by `select_one` is $\langle \texttt{prune}, \texttt{nothing} \rangle$. The sequence obtained by the parallel composition is $\langle p_3 || \texttt{nothing} \rangle$, and thus by simplification $\langle p_3 \rangle$.

In synchronous languages, "don't-care nondeterminism" can be implemented through *oracle* variables in Esterel [Tec05] and various nondeterministic statements like `choose` in Quartz [Sch09]. These nondeterministic features are used to verify the behavior of a synchronous program under a test environment.

Chapter 7

Interactive Computer-Aided Composition

Computer-aided composition systems enable composers to write programs to generate and transform musical scores. For this purpose, the paradigm of constraint programming is appealing due to its declarative nature: the composer constrains the score and relies on a constraint solver to propose solutions. However, the existing tools lack interactivity, and composers do not participate in the selection of a particular solution. In this chapter, we propose a score editor in which the composer can navigate in the solution space to select a solution of its choice. We implement an interactive search strategy, in spacetime programming, for lazily building the solution space of a problem. It demonstrates the usefulness of the synchronous aspect of spacetime to interactively solve a constraint problem. This chapter is an edited and extended version of the work appearing in [TAE17].

7.1 Introduction

Computer-aided composition is a routine for many composers, as attested by numerous tools including OpenMusic [AADR98] and Max/MSP [PZ90]. It enables the composer to delegate tedious computation to the machine, such as generating rhythms for non-overlapping voices of a score. The computation is usually displayed in visual programming languages based on the functional paradigm. In this paradigm, the data "flows" in a tree structure where nodes (named "boxes") encapsulate computation on data. If a functionality is missing in the available precoded boxes, the composer must implement it with a "lower-level" programming language, such as Lisp in OpenMusic. However, these programming languages are less intuitive for composers than visual languages. This is why other paradigms, such as constraint programming, are investigated.

Constraint programming has been applied to model multiple aspects of music theory, such as harmony, rhythm and orchestration [TA11]. There are several systems integrating constraint programming in computer-aided composition softwares [AM11]. In particular, PWConstraint [Lau96b] is one of the first systems that integrates constraint solving under a visual composition environment. Another approach is OMCloud [TC04] that is based on a non-exhaustive constraint

solving technique called *local search*. Generally, merging the constraint and functional paradigms is done by encapsulating constraint solving into a box where the parameters are inputs to the constraint satisfaction problem (CSP) and the output is a solution to this CSP.

Lack of interactivity

A CSP can have from zero (in case of unsatisfiability) to multiple solutions. Despite the relational nature of constraints, the existing systems view a CSP as a function. Therefore, the solution chosen by the solver is unpredictable and the composer does not participate in the selection of this particular solution. Besides, this process is not replicable: the first solution may change with the solver's internal search strategy, parameters or when the solver is simply updated.

On the contrary, a CSP can be over-constrained with no solution. In this case, a common method is to use soft constraints: the system tries to satisfy as many constraints as possible. It is similar to the problem with multiple solutions because many "soft solutions" are possible. In summary, current approaches lack interactivity between the composer and the constraint solver for selecting the solution.

Interactive score editor

To solve this problem, we suggest a score editor in which the composer can visualize partially instantiated scores and steer the solving process toward a customized solution (Section 7.2). We propose several interactive strategies for navigating in the solution space to help the composer consciously select a solution (Section 7.3). However, the existing abstractions inside constraint solvers are not tailored for interactivity. This is why we are using the spacetime paradigm to facilitate interactions between the composer and the solver. The synchronous part of spacetime is the key to support interactive solving. We experiment the system with the all-interval series problem (Section 7.2) and the diagnostic of musical rules (Section 7.3.3). In the latter, instead of using soft constraints, we dynamically alert the composer when we detect the violation of a rule. The result is an interactive score editor with constraint solving as a part of the composition process.

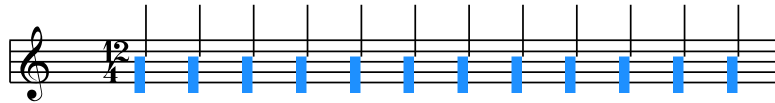
7.2 Score editor with constraints

7.2.1 Visual constraint solving

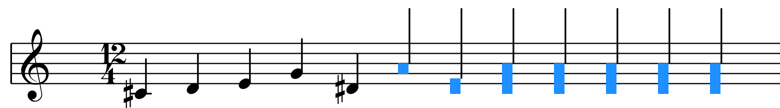
We propose a new score editor programmed in Java. We particularly focus on the visual and interactive aspect of constraint solving. To illustrate the system, we use the all-interval series (AIS) musical constraint problem. It constrains the pitches to be all different as well as the intervals between two successive pitches. This is notably used to implement the twelve-tone technique in which every note of a

pitch class has the same importance. This constraint comes built-in in our system and we leave apart its exact modeling which is covered in Section 1.1.1.

Initially, when the AIS problem is set in the editor, the pitches are initialized with domains in the interval [1..12] and are represented with rectangles:



Through the solving process, these rectangles become smaller and are displayed as a note when instantiated. For example, the following score is partially instantiated with four notes and the propagation reduces the domains of the rest of the score accordingly—the rectangles became smaller:



Experimentally, the 'space' key is pressed until a partial solution or a fully instantiated solution satisfies the composer. An example of solution given by our system to the all-interval series is:

These scores are displayed in a larger visual programming environment similar to OpenMusic. In this setting, a score is contained in a functional box which ensures the compatibility with existing methods.

7.2.2 Spacetime for composition

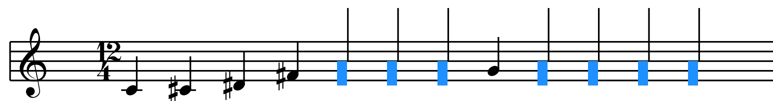
The model of the problem can be monotonically updated (adding constraints) throughout the solving process. Between instants, the composer is allowed to add new constraints into the model. We identify two different ways to add constraints interactively based on the spacetime specifiers:

- *Persistent*: The constraints are added in a store with the spacetime `single_space`, and they hold for the rest of the search. For example, a chord that the composer particularly likes and wants to be part of the final musical composition.
- *Contextual*: The constraints are added in a store with the spacetime `world_line`, and they hold only for the current subtree. For example, it can be an interval between two notes that only makes sense in the presence of the already instantiated notes.

To achieve this, we need to modify the solver presented in Chapter 3. We only highlight the changes here:

```
class PSolver {
  world_line CStore constraints = bot;
  single_space CStore cpersistent = bot;
  flow merge_cstore = constraints <- cpersistent;
}
```

We use an additional constraint store `cpersistent` that can be augmented by user constraints in between instants. In each instant, we impose these constraints in the initial `constraints` store, hence they are never "forgotten". For example, here, the composer interactively chooses to instantiate the eighth note to G:



G is added in the persistent constraint store, and will remain unchanged until the end of the search. Hence, every partial assignment or solution will contain this note.

7.3 Interactive search strategies

Using the spacetime paradigm, we investigate several search strategies from the most straightforward to the more complex but useful strategies.

7.3.1 Stop and resume the search

There are many ways to interact with a search tree during its traversal. Interacting in each node is not really interesting because the search tree is usually too large and we are not interested in every partial assignment. In most composition-aided systems, the user interacts with the search at solution nodes and, if needed, asks for the next solution. This behavior is programmed in spacetime with the following code:

```
loop
  single_time ES consistent = read constraints.consistent(read domains);
  when consistent == true then
    stop
  else
    pause
  end
end
```

The statement `stop` gives the control back to the host program when we reach a solution's node.

More generally, we can stop the search on any event. For example, we can be interested by a partial assignment in which a new variable has just been instantiated:

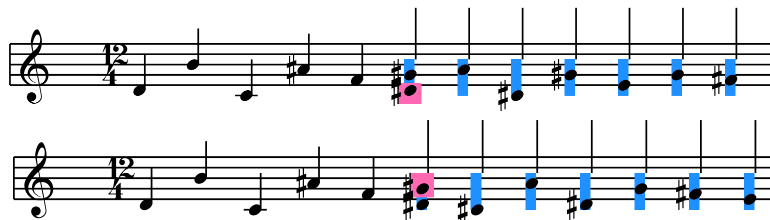
```
world_line LMax asn = new LMax(0);
loop
  asn <- count_asn(read domains);
  when pre asn |< asn then
    stop
  else
    pause
  end
end
```

The variable `asn` represents the number of variables instantiated in `domains`. It is of type `LMax` with a `world_line` specifier because the number of assignments can only increase along a path of the search tree. In each node, we update this value by calling `count_asn()` on `domains`. We suspend the search whenever the current number of assignments is greater than the previous one.

7.3.2 Lazy search tree

A musical CSP can have many solutions, especially in the early composition of the musical piece because it is under-constrained. The set of solutions, proposed by the strategies presented above, is a catalogue in which the composer picks one solution. However, their analysis by the composer is not practical and computing every solution can be time-consuming. We propose a search strategy interleaving solution generation and composer interaction. The goal is to obtain a solution that has been entirely chosen by the composer, but without exploring the full solution space.

We call it a lazy search strategy because it explores the solutions space *on-demand*. This strategy explores the score from left to right, and whenever a note can be instantiated to several pitches, the composer chooses one. For example, the next two scores represent a choice between ♯*D* and ♯*G* on the sixth note—framed with a red rectangle:



The strategy first computes a representative solution for each ♯*D* and ♯*G*. It is mandatory if we want the composer to navigate in the solution space and not the

full search space. To summarize, each time two or more solutions exist in a given note's domain, the system performs the following tasks:

(i) it pauses the search strategy,

(ii) it asks the composer for the note wanted, and

(iii) it discards all the other propositions and resumes the search.

The laziness comes from the fact that the search tree of the discarded solution will not be explored further. We present the algorithm in Figure 7.1 and we comment the main points of this algorithm.

Firstly, the key point of this algorithm is to rely on two queues `StackLR left` and `StackRL right` for respectively exploring the tree from left to right and right to left. We create one universe for each queue and solve the CSP from both directions (process `solve`). Every time we reach a solution, the universe is paused up (process `pause_up_on_solution`). Interestingly, thanks to the fact that we cannot observe changes on `world_line` variables in lower universes, we can use the model for both search strategies.

Secondly, the variable `choice` reflects the decision of the composer to explore the left or the right part of the solution space. Its type is `L<Boolean>` where `L<T>` is a Java class transforming any type `T` into a flat lattice. We ask the choice of the composer in the process `commit_composer_choice`, and more precisely through a call to the host function `ask_composer`. The host language can display the score on a graphical interface because we pass the domains of both the left and right parts in arguments. We also retrieve potential constraints from the composer that are added persistently into the root node of the CSP. If we want to add a contextual constraint, we can do so with a write on `left::constraints` for example.

Thirdly, alternating between the left to right and right to left strategies is done in the process `search`. In the first instant, we need to explore both solutions, and we rely on the process `init` to explore both part. Interestingly, we cannot put these processes in parallel because they might not explore the same number of nodes, and thus they will not be fully synchronized. Afterwards, we explore either the left or right part of the search tree depending on the composer choice. This is implemented with the `suspend` statement which activates the left search strategy if the composer has kept the solution of the right part, and vice versa.

We detect that the full search tree has been explored when the two search strategies find the same solution. In this case we terminate the search. The other termination condition is when the CSP is unsatisfiable.

7.3.3 Diagnostic of musical rules

The lazy search algorithm is especially useful in case of under-constrained problems with many solutions, but it does not help if the CSP is unsatisfiable in the first place. Current approaches [AM11] use soft constraints: a rank is given to

each constraint and it tries to maximize the number of constraints satisfied. For example, it can be used to define the rules of the species counterpoint. It defines many rules for composing and some of them can be violated. However, it is not easy to define preferences among the different rules. We take another approach where the rules are "observer processes" detecting whenever they are unsatisfied:

```
class RuleDetection {
  world_line VStore domains;
  world_line CStore rule;
  single_time ES entailed;

  public RuleDetection(VStore domains, CStore rule) { ... }

  flow detect = entailed <- read rule.consistent(read domains);
}
```

The class `RuleDetection` is composed of the variable `domains` of the current CSP and of the variable `rule` which is a constraint store containing the rule to monitor. The process `detect` updates the variable `entailed` in each instant to reflect the status of the current counterpoint rule relevant to the current search tree. When it is equal to `false`, we know the current rule to be unsatisfiable and we can inform the composer about it. Using the parallel operator, we can monitor the status of any number of rules and they can be composed with any of the strategies presented above.

7.4 Conclusion

Computer-aided composition with constraints is not often used due to the black box search process in constraint solvers. We introduced a score editor with an interactive search strategy allowing to navigate in the solution space. Hence, the composer knows clearly why a solution is chosen. With the spacetime paradigm, we are able to lazily explore the search tree, to pause and to resume the search with additional information from the composer. In addition, at any stage of the search, the partial solution can be visualized on the score and examples of possible solutions are given. Lastly, we show that without solving a CSP, we can still check if some constraints become unsatisfiable during the composition process and alert the composer about it.

The modeling of musical constraint problems has been left apart. In the future, we want to incorporate visual modeling capabilities in our score editor that fits the interactivity of the search well. Last but not least, we will evaluate and experiment this editor with professional composers.

```
class LazySearch {
  single_space VStore domains;
  single_space CStore constraints;
  single_time L<Boolean> choice = bot;

  single_space StackLR left = new StackLR();
  single_space StackRL right = new StackRL();

  LazySearch(VStore domains, CStore constraints) { ... }

  proc solve(q) =
    universe(world_line domains, world_line constraints) with q in
      Solver solver = new Solver(domains, constraints);
      par
      || solver.search();
      || pause_up_on_solution(solver);
      end
    end

  proc pause_up_on_solution(solver) =
    loop
      when solver.consistent |= true then
        pause up
      else
        pause
      end
    end

  flow commit_composer_choice =
    single_time CStore c =
      ask_composer(read left::domains, read right::domains, write choice);
    constraints <- c;

  proc init = solve(right); pause; solve(left);

  proc search =
    init();
    pause;
    weak abort when left::domains == right::domains or
                    left::domains |= false in
      par
      || suspend when choice |= true in solve(right) end
      || suspend when choice |= false in solve(left) end
      || commit_composer_choice()
      end
    end
}
```

Figure 7.1: Lazy search algorithm.

Chapter 8

Model Checking with Constraints

All along this work, we illustrated spacetime programming with search strategies for solving constraint satisfaction problems. In this chapter, we broaden the application scope of spacetime to model checking. Model checking is a verification technique to check the satisfiability of a formula on an abstraction of a system, called a model. Compared to constraint programming, the state space generated by the model is non-monotonic and dynamically built during the exploration. After formally describing model checking, we cast the definition of the model into a lattice that can be manipulated within the spacetime paradigm. For this purpose, we use a constraint representation of the model's states. The search strategy, solving the CSP associated to the model, is composed with the model exploration and is crucial to detect unreachable states. Afterwards, we show that a logic formula can be viewed as a high-level search strategy that can itself be expressed in spacetime. All in all, we propose a spacetime program combining the model checking search algorithm with the constraint solving procedure and the logic formula to verify. *This chapter is the current status of an on-going collaboration with Clément Poncelet, and a first version has been published in [TP17].*

8.1 Model checking

Model checking [BK08] is a verification technique which, given a model $\mathcal{M}$, aims to establish its conformance on a given formula F. For this purpose, it requires a specification of the system to test, represented with abstract graphs, and of the property to satisfy, usually described in a temporal logic. From these specifications, a verification procedure is used to explore the state space induced by the model $\mathcal{M}$ and to verify that the logic formula F is satisfied for each state $s \in \mathcal{M}$. The formula is specified with a set of atomic propositions AP, and a labelling function $\mathcal{L}$ specifying the atomic propositions valid in each model state. Examples of atomic propositions include state labels such as "the current state is equal to the label ℓ_1", and some constraints to satisfy in a state such as $x > 0$ where x is a variable of the model. Overall, the verification procedure succeeds if it can assess the model

conformance to the formula, otherwise it returns a counterexample to guide the user from an initial state of $\mathcal{M}$ to an erroneous one s_{error} violating the considered property.

The search algorithms are central to such verification procedures and are crucial for the efficiency of model checkers. Modern model checkers combine several search strategies in order to reduce the complexity in time and space [CGK+13] and to obtain better counterexamples [HF11, QW04].

Formally, a model $\mathcal{M}$ is defined as a set of program graphs modelling the control flow graph of a program and the processes synchronization. The program graphs are defined on the same set of actions Act, statements $Stmt$ and variables Var. Given $\mathsf{Cond}(Var)$ a set of boolean conditions on Var and $\mathsf{Eval}(Var)$ the evaluation function of the variables, each program graph is a tuple $PG_i = \langle Loc, \mathsf{Effect}, \hookrightarrow, Loc_0, g_0\rangle_i$ where:

(i) Loc is a set of locations,

(ii) $\mathsf{Effect} : Stmt \times \mathsf{Eval}(Var) \rightarrow \mathsf{Eval}(Var)$ is a set of effect functions,

(iii) $\hookrightarrow: Loc \times \mathsf{Cond}(Var) \times Act \times Stmt \times Loc$ is a set of transitions,

(iv) $Loc_0 \subseteq Loc$ is the set of initial locations, and

(v) $g_0 \in \mathsf{Cond}(Var)$ is the set of initial conditions on PG_i variables.

We denote the transition $\langle \ell, g, a, \alpha, \ell' \rangle \in \hookrightarrow$ as $\ell \xrightarrow{g:a:\alpha} \ell'$ where ℓ is the source location, ℓ' is the target location of a transition, g is its guard (the transition's firing condition), a is the communication action, and α is the effect.

The semantics of a set of n program graphs $PG_{1..n}$ is defined as a transition system TS specifying the execution of n processes $PG_1 \| \ldots \| PG_n$ in parallel. A transition system TS is defined with $\langle \vec{S}, Act, \rightarrow, \vec{I}, AP, L\rangle$ where:

(i) $\vec{S} = \langle s_1, \ldots, s_n\rangle$ with $s_i : Loc_i \times \mathsf{Eval}(Var)$ a pair of a program graph location and its variables values.

(ii) $Act = \bigcup_{i \in [1 \ldots n]} Act_i \cup \tau$ is the union set of the program graph actions with the τ-action abstracting the internal system transitions.

(iii) $\rightarrow: \vec{S} \times Act \times \vec{S}$ is the transition in the system as defined below.

(iv) $\vec{I} = \langle I_1, \ldots, I_n\rangle$ is the initial states vector, that is, for each program, a pair of initial locations and variables values satisfying its initial conditions.

(v) $AP = \bigcup AP_i$ is the set of atomic propositions of the problem.

(vi) $L(\langle \ell, \eta\rangle) = \{\ell\} \cup \{g \in \mathsf{Cond}(Var) \mid \eta \models g\}$ is the labelling function mapping a state $\langle \ell, \eta\rangle \in \vec{S}$ to the atomic propositions satisfied by this state.

The transition $\rightarrow$ specifies how the global system TS progresses from a state $\vec{S}$ to another state $\vec{S}'$. The following semantics rules describe the internal transition and the synchronization of two transitions on an action $a \in Act$:

INTERNAL

$$\frac{s_i = \langle \ell_i, \eta_i \rangle \xrightarrow{g:\tau:\alpha}_i \langle \ell_i', \eta_i' \rangle = s_i' \qquad \eta_i \models g \qquad \eta_i' = \mathsf{Effect}(\alpha, \eta_i)}{\langle s_1, \ldots, s_i, \ldots, s_n \rangle \xrightarrow{\tau} \langle s_1, \ldots, s_i', \ldots, s_n \rangle}$$

SYNCHRONIZATION

$$\frac{\begin{array}{c} s_i = \langle \ell_i, \eta_i \rangle \xrightarrow{g_i:a?:\alpha_i}_i \langle \ell_i', \eta_i' \rangle = s_i' \qquad \eta_i \models g_i \qquad \eta_i' = \mathsf{Effect}(\alpha, \eta_i) \\ s_j = \langle \ell_j, \eta_j \rangle \xrightarrow{g_j:a!:\alpha_j}_j \langle \ell_j', \eta_j' \rangle = s_j' \qquad \eta_j \models g_j \qquad \eta_j' = \mathsf{Effect}(\alpha, \eta_j) \end{array}}{\langle s_1, \ldots, s_i, \ldots, s_j, \ldots, s_n \rangle \xrightarrow{a} \langle s_1, \ldots, s_i', \ldots, s_j', \ldots, s_n \rangle}$$

We define the function

$$\mathsf{post}(\vec{S}) = \{\ell \xrightarrow{g:a:\alpha} \ell' \mid \langle \ell, \eta \rangle \in \vec{S} \ \wedge \ \langle \ell, \eta, g, a, \alpha, \ell', \eta' \rangle \in\rightarrow\}$$

which maps the state vector $\vec{S}$ to a set of enabled program graph's transitions.

We bring all the definitions together. Model checking consists in answering if the logic formula Φ is valid in the transition system ts, which is written $ts \models \Phi$. The two foundational temporal logics are the linear temporal logic (LTL) and the computation tree logic (CTL). A standard technique [BK08] is to transform the negation of the logic formula into a Büchi automaton reading the words—made of atomic propositions—accepted by the formula. Hence, if the intersection of the transition system and the formula is empty, it means that the formula is unsatisfiable, and thus valid in the transition system. Finally, we denote Sat the set of states satisfying a logic formula. To sum up, a model assesses a property Φ if $\exists i \in I$ such that $i \models \Phi$, *i.e.* at least one initial state of TS satisfies Φ.

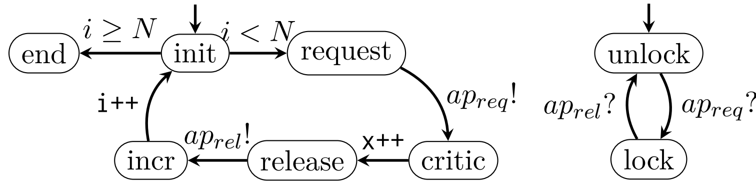


Figure 8.1: Two program graphs specifying the Peterson mutual exclusion problem.

Example 8.1. As an example, we define the Peterson mutual exclusion problem for which the model is depicted in Figure 8.1. On the left, the first program graph models the critical access of a global variable x, and a program incrementing x until its local variable i equals N. Its set of initial locations Loc_0 is $\{init\}$ (specified

by the arrow without source node), its variables $Var = \{i, x\}$ and its actions $Act = \{ap_{req}, ap_{rel}\}$. For clarity, we did not assign both a synchronization and an effect on transitions, although it would be correct. In addition, τ-actions, empty effect—denoted ϵ—and *true* conditions are omitted.

We can instantiate the left and right models an arbitrary number of times. Each instantiation is a program graph executed as a process. For instance, we can instantiate the left program graph twice, noted P_1 and P_2, and the right one once, noted L. Assuming the variables initialized to 0, we have the initial system state $\vec{S_0} = \{\langle P_1.init, \{0,0\}\rangle, \langle P_2.init, \{0,0\}\rangle, \langle L_1.unlock, \{0\}\rangle\}$, with $\{0,0\}$ meaning that, for $k \in \{1,2\}$, we have $\mathsf{Eval}(P_k.i) = 0$ and $\mathsf{Eval}(x) = 0$. From this initial state, the only two enabled transitions are the ones targeting the request location for each program graph P_1 and P_2, formally defined as:

$$\mathsf{post}(\vec{S_0}) = \{P_1.init \xrightarrow{P_1.i<N:\tau:\epsilon} P_1.request, P_2.init \xrightarrow{P_2.i<N:\tau:\epsilon} P_2.request\}$$

⌟

8.2 Model checking with constraints

In order to combine constraint solving and model checking, we use lattices as a common ground to represent their state spaces. This combination allows us to tackle several limits of these two paradigms by mixing their different approaches. On the one hand, constraints are useful for pruning large subtrees by detecting incoherences as soon as possible using propagation. However, bounded model checking usually takes a "generate-and-test" approach enumerating the possible values of the variables and choices of branches during a formula verification. On the other hand, constraint programming is not suited for dynamically building a CSP during the exploration whereas it is at the core of model checking algorithms.

To combine these two approaches, we propose to view the guards and effects in model checking as constraints that are dynamically accumulated when exploring the model's state space. For this purpose, we first merge the definitions into a lattice framework that is later used in Section 8.3 to program the search algorithm.

8.2.1 Constrained transition system

We first establish equivalences between some of the terminologies employed in both fields. Model checking defines the set of variables Var, the evaluation function $\mathsf{Eval}(Var)$ (denoted η) and Effect for modifying the assignments of the variables. Interestingly, a CSP is defined over $\langle d, P\rangle \in L_3$ where the set of locations in d is the set of variables, d is the evaluation function, and the propagators P are the effects over the variables.

The major step in the merging of both definitions is to lift the evaluation function $\mathsf{Eval}(Var)$ to variables defined on domains rather than on single values. Since variables are induced by constraints, the evaluation function handled into

transitions is replaced with a CSP $\langle d, P\rangle \in \mathsf{L}_3$. We refine the definition of program graph to a constrained program graph which is a tuple $PG_i = \langle Loc, \mathsf{Effect}, \hookrightarrow, Loc_0, g_0\rangle_i$ where:

(i) Loc is a set of locations.

(ii) $\mathsf{Effect} : Stmt \times L_3 \to L_3$ is a set of effect functions adding a constraint to a CSP.

(iii) $\hookrightarrow: Loc \times Prop \times Act \times Stmt \times Loc$ is a set of transitions.

(iv) $Loc_0 \subseteq Loc$ is the set of initial locations.

(v) $g_0 \in Prop$ is the set of initial conditions on PG_i variables.

The guards and effects of a transition are both elements of the propagators' set $Prop$. Moreover, the constrained transition system TS is defined with $\langle S, Act, \to, \vec{I}\rangle$ where:

(i) $S = \vec{Loc} \times L_3$ is the current state.

(ii) $Act = \bigcup_{i\in[1\ldots n]} Act_i \cup \tau$ is the set of program graph actions (see Section 8.1).

(iii) $\to: S \times Act \times S$ are the two possible kinds of transitions.

(iv) $\vec{I} = \langle I_1, \ldots, I_n\rangle$ is the initial states vector.

Since the whole model is defined over a constraint system, the set of atomic propositions AP and the labelling function L are removed from the transition system. Indeed, these items are used to check if a given location assesses an atomic proposition and this verification is handled by our CSP during the formula verification. Instead, an atomic proposition is an arbitrary constraint $[\![c]\!] \in Prop$ and the labelling function is the entailment relation $\models_3$.

The semantics rules of the transition $\to$ must be redefined with a CSP contained in the state:

INTERNAL

$$\frac{\langle \ell_i, \langle d, P\rangle\rangle \xrightarrow{g:\tau:\alpha}_i \langle \ell_i', \langle d', P'\rangle\rangle \qquad \langle d', P'\rangle = \mathsf{Effect}(\alpha, \langle d, P \wedge [\![g]\!]\rangle) \qquad sol(\langle d', P'\rangle) \neq \emptyset}{\langle\langle \ell_1, \ldots, \ell_i, \ldots, \ell_n\rangle, \langle d, P\rangle\rangle \xrightarrow{\tau} \langle\langle \ell_1, \ldots, \ell_i', \ldots, \ell_n\rangle, \langle d', P'\rangle\rangle}$$

SYNCHRONIZATION

$$\frac{\begin{array}{c}\langle \ell_i, \langle d, P\rangle\rangle \xrightarrow{g_i:a?:\alpha_i}_i \langle \ell_i', \langle d', P'\rangle\rangle \qquad \langle \ell_j, \langle d, P\rangle\rangle \xrightarrow{g_j:a!:\alpha_j}_j \langle \ell_j', \langle d', P'\rangle\rangle \\ \langle d', P'\rangle = \mathsf{Effect}(\alpha_i;\alpha_j, \langle d, P \wedge [\![g_i]\!] \wedge [\![g_j]\!]\rangle) \qquad sol(\langle d', P'\rangle) \neq \emptyset\end{array}}{\langle\langle \ell_1, \ldots, \ell_i, \ldots, \ell_j, \ldots, \ell_n\rangle, \langle d, P\rangle\rangle \xrightarrow{a} \langle\langle \ell_1, \ldots, \ell_i', \ldots, \ell_j', \ldots, \ell_n\rangle, \langle d', P'\rangle\rangle}$$

The function Effect is used to obtain a new CSP after we applied the effects and the guards of the transition. This new CSP must at least have one solution.

In particular, if the CSP has one solution, it means that the guards are entailed by the current CSP and that we can proceed. We see in Section 8.4.2 how we compute this entailment relation. Last, we define the post function:

$$\mathsf{post}(\langle \vec{\ell}, \langle d, P\rangle\rangle) = \{\ell \xrightarrow{g:a:\alpha} \ell' \mid \ell \in \vec{\ell} \wedge \langle \ell, \langle d, P\rangle, g, a, \alpha, \ell', \langle d', P'\rangle\rangle \in \rightarrow\}$$

which maps the state $\langle \vec{\ell}, \langle d, P\rangle\rangle$ to a set of enabled program graph's transitions.

8.2.2 Existential constraint system

The Effect function translates a statement $s \in \mathit{Stmt}$ into a propagator $p \in \mathit{Prop}$ that is then added to the current CSP. However, the translation of an assignment into a constraint is not direct and needs a special treatment. For example, the effect statement $x := x+3$ is not a constraint since the assignment $:=$ is a function, not a relation, and thus the constraint $x = x+3$ is unsatisfiable. In existing approaches, this problem is solved by giving a stream interpretation to the variables [BTM11, LLLS11, LL14]. The variables are coupled with a timestamp and form a sequence of values $(v_1, v_2, \dots, v_n)$ such that v_i is a variable at the time point i. In this setting, the previous example is transformed into the constraint $x_{t+1} = x_t + 3$. Constraints on streams involve non-trivial changes in the constraint solvers because: (i) streams are infinite sequence, and thus the standard search algorithm—operating on finite domains—needs to be adapted, and (ii) constraints must be lifted to operate on streams instead of domains.

We observe that guards and effects in a constrained transition system always refer to the last assigned value. Therefore, at any point in time t, the past values at time $t_i, i > 0$ can be hidden since new constraints will not involve them. We endow the constraint system with an existential operator $\exists$ allowing to declare fresh variables in a CSP. This operator is taken from the concurrent constraint programming paradigm [SR89] for locally declaring new variables associated with a constraint system.

We assume that the domain d is defined over the set of names Name. The existential operator is implemented with a substitution operator over the CSP such that, for a fresh variable $y \in \mathit{Name}$, $\exists x.\langle d, P\rangle \stackrel{\text{def}}{=} \langle d, P\rangle[x \to y]$. The substitution $\langle d, P\rangle[x \to y]$ replaces every occurrence of x in d and P by the variable named y. For example, given an arithmetic expression expr, we transform an assignment $x := \mathit{expr}$ to a constraint $\exists z.(z = \mathit{expr} \wedge \exists x.x = z)$ where $z \in \mathit{Name}$ is fresh in the constraint system. The variable z serves to evaluate the expression expr with the past value of x. We hide the variable x just after and assigns z to the newly declared x. Using the substitution operator only, we can also define the function $\mathsf{Effect}(x := \mathit{expr}, \langle d, P\rangle)$ as follows:

$$\langle d', P'\rangle = \langle d, P\rangle[x \to y] \tag{8.1}$$

$$\langle d' \sqcup_2 (x, d(x)), P' \sqcup_3 [\![x = \mathit{expr}[x \to y]]\!]\rangle \tag{8.2}$$

First, we substitute the variable x in the current CSP with a fresh variable y (equation 8.1), and second, we add the assignment constraint $x = expr[x \rightarrow y]$ in the new CSP (equation 8.2). In the previous example, the assignment $x := x + 3$ is transformed into $x = y + 3$ where y refers to x before the substitution. The notion of stream still implicitly exists but is hidden in the constraint system and we cannot constrain the evolution of a variable through time (such as with stream constraints).

The existential quantifier glues CSP solving with the dynamic creation of the model. Therefore, we can check constraint such as $x = 1$ in every state even if x does not evolve monotonically (Section 8.4).

8.2.3 Lattice abstraction

Model checking as well as constraint solving are based on a backtracking algorithm for exploring a state space. The major difference between both formalisms is that the state space in model checking is infinite while it is finite in constraint programming. In this section, we present a lattice abstraction of a state of the constrained model. This abstraction is then used in Section 8.3 to develop the full model checking algorithm.

The lattice of a constrained transition system is infinite because the existential operator extends the set of variables when a cycle is encountered. Formally, the state space is a set $Loc^n \times L_3$ where $\vec{\ell} \in Loc^n$ is the set of locations of the n program's graphs, and $\langle d, P\rangle \in L_3$ is the current constraint system. The order of this lattice $\langle \vec{\ell'}, \langle d', P'\rangle\rangle \models \langle \vec{\ell}, \langle d, P\rangle\rangle$ is defined if $\langle d', P'\rangle \models \langle d, P\rangle \wedge \vec{\ell} \rightarrow^* \vec{\ell'}$. A state s is "after" another state s' if we can reach the location of s from s', and if the information presents in s' can be deduced from s. We detect an already visited state when we reach a state with the same location and in which we did not gain any information.

Step A lattice's transition is a "step" in the model checking process. Given the function post and with m the sum of enabled transitions, every step has m possible next transitions. Indeed when several next transitions are possible, the choice is nondeterministic in model checking. For our CSP, a step implies:

(i) the computation of these m transitions from $\vec{S}$ (notice that it can be higher than the number of program graphs for the first step since a PG may have more than one initial location), and

(ii) for each transition $\ell_i \xrightarrow{g:a:\alpha} \ell'_i \in \mathsf{post}(\vec{S})$:

  (a) Joining the guard g in P,

  (b) Checking with the new assignment a of x_{i+1}, *i.e* if $a \in Sol(\langle d, P\rangle)$, and the storage of a in the new assignment x_{i+1} for each variables x_{i+1} in α.

$$P' = P \cup [\![g]\!] \cup [\![\alpha]\!].$$

(iii) the computation of the next states $\vec{S}'$ (for each transition) with the new lattice $\langle\langle d', P'\rangle, \vec{\ell}'\rangle$.

Remark: We mimic the application of a model checker step, following the relation $\rightarrow$ defined Section 8.1. A transition application can be one τ-transition and only one transition is taken, or a synchronization transition which implies two transitions firing: the action emission ($a!$) and reception ($a?$). In this case, the guard added to P is $g = g_1 \wedge g_2$ and the effects $\alpha = \alpha_1 \wedge \alpha_2$. Moreover, once an emission is received, every complementary action ($a?$) in a different source can be fired non-deterministically, this should be managed in the CSP.

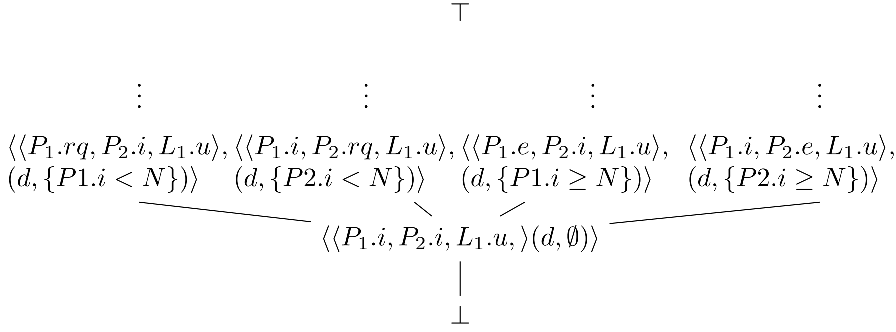


Figure 8.2: Lattices for *TS* Perterson example.

Example 8.2. We depict the lattice of the Peterson mutual exclusion problem in Figure 8.2. For a lattice node, the first item contains the control locations of each program graph, and we denote $P.i$ for $P.init$, $P.rq$ for $P.request$, $P.e$ for $P.end$ and $L.u$ for $L.unlock$. The second item contains the variables domains and their constraints applied when a transition is fired. The CSP is initialized with the initial constraints g_0 and the variable's domains (the local variable i for each process P are denoted respectively $P1.i$ and $P2.i$). In this example, we have the variables of d be {x,P1.i,P2.i} and the constraint store is empty since we do not have initial constraints. The first step (the result of `post`) generates 4 transitions, 2 for each program graph P_i, being the two branches of the if statement. The lattice elements resulting from this step are the four possible choices, each with a fired transition, and such that they added their constraints into the CSP. ⌟

Based on the definitions of Section 8.2 we describe dynamic creation and exploration of the state space. To this aim, we use the spacetime language and a library representing the abstract transition system *TS* which provides the methods:

post returning the set of the enabled transitions from the current $\vec{S}$ following $\rightarrow$. This library does not handle constraints management but only the synchronization feature.

apply requiring a transition and applying it onto the current state. Two behaviors are possible:

- Nothing is returned, and the method updates the current state with the input transition's target.
- A set of transitions is returned. They are the enabled transition synchronized with the action-transition applied in input.

8.3 Spacetime algorithm

In Section 8.2, we viewed the CSP as a lattice object that evolves when we pass a transition. Specifically, in each state, we check the condition $sol(\langle d', P' \rangle) \neq \emptyset$ that ensures the reachability of the current state, i.e. the CSP $\langle d', P' \rangle$ must have at least one solution. However, establishing the satisfiability of a CSP is costly: it generates its own state space and can require complex algorithms to be solved efficiently. Hence, the search algorithm of a constrained transition system induces two layers of non-determinism: (i) at the transition level in the model, and (ii) at the CSP level. Using the spacetime language, we provide the description of an algorithm interleaving transition execution and constraint solving. Also, a major optimisation consists in resuming the solver from the previous solution computed instead of restarting everything from scratch.

Our search algorithm is implemented in two classes: `StateSpace` implementing the state space of the transition system and `CSP` for encapsulating the constraint solving.

```
class StateSpace {
  world_line VStore domains;
  world_line CStore constraints;
  world_line TransitionSystem ts;
  world_line StackLR queue;

  public StateSpace(VStore domains, CStore constraints,
    TransitionSystem ts, StackLR queue) { ... }

  public proc search =
    par
    || next_transition();
    || prune_unreachable();
    || solve();
    end

  flow next_transition =
    for(;)(single_time Transition t: readwrite ts.post())
      space readwrite ts.apply(read t, write domains, write constraints) end
    end
```

```
  flow prune_unreachable =
    single_time ES consistent = read queue::constraints.consistent(read queue::domains);
    when consistent == false then
      prune;
    end

  proc solve =
    universe(world_line domains, world_line constraints) with queue in
      Solver solver = new Solver(domains, constraints);
      par
      || solver.search();
      || pause_up_on_solution(solver);
      end
    end

  proc pause_up_on_solution(solver) =
    loop
      when solver.consistent |= true then
        pause up
      else
        pause
      end
    end
}
```

The field `ts` instantiates the constrained transition system TS as defined Section 8.2.1, and holds the set of current locations $\vec{\ell}$. These fields form the state $\langle \vec{l}, \langle d, P \rangle \rangle$ of the transition system as defined in Section 8.2.3. We build the set of future states to explore in `next_transition`. For this purpose, two methods on `TransitionSystem` are provided:

(i) `post` returns the set of the possible next transitions given by $\mathsf{post}(\langle \vec{\ell}, \langle d, P \rangle \rangle)$.

(ii) `apply` commits to a transition in the system by updating the set of locations and applying the guards and effects to the current CSP, possibly with variable substitutions (Section 8.2.2).

The process `prune_unreachable` verifies in each instant whether the CSP holds a solution or is failed. In the latter case, it means that the search process was not able to find a solution and that the current state is unreachable. Hence, we do not further explore the rest of the current state space and backtrack using the statement `prune`. Note that the CSP is also backtracked.

An important aspect is that the queue of nodes has the specifier `world_line`. It means that whenever the transition system is backtracked, we also backtracked the queue of nodes of our CSP. Therefore, we restore our system to the exact same solving state as before, without recomputing a solution.

8.4 Search strategies for verification

Models are verified with formula in some temporal logic such as *linear temporal logic (LTL)* or *computation tree logic (CTL)*. They express how the state space is explored in an abstract way. We can view these logic formulas as pruning strategies specifying only the part of the state space they are interested in.

In model checking, temporal logic is commonly used to express properties that must be verified on the model (Section 8.1). Instead of relying on a specific temporal logic, we directly specify our property with spacetime. Although being more verbose than a logic formula, the advantage of spacetime is to make accessible the concepts of state space creation, satisfiability checking of the formula and search strategies to optimize the exploration. Another advantage is that a spacetime program describes an executable algorithm. Nevertheless, comparing logic formulas with the spacetime paradigm is an interesting research topic that we reserve for future work.

In this section, we study the satisfiability checking of properties in the transition system. Firstly, we provide two examples where the properties to check can be concisely expressed using constraints. Secondly, we describe an entailment algorithm for computing $ts \models \Phi$ where Φ is a constraint instead of a formula.

8.4.1 Verifying constraint-based property

We show two examples where constraint programming is useful for proving structural properties. For this purpose, we use global constraints to capture the substructures of a problem. There is a substantial amount of global constraints [BCR11] covering different aspects of constraint modelling such as scheduling, packing or sorting.

Example 8.3 (at_most)**.** In the Example 8.1, we modelled the Peterson mutual exclusion problem. It models processes sharing a single resource x that must be accessed by one process at a time. This property can be captured by the *at_most(n, array, v)* counting global constraint. It enforces that at most n variables take the value v in *array*. In the Peterson example, we want at most one process in the state $release_i$ and $critic_i$ at the same time. We write this constraint as $at_most(1, \forall i \in N.[critic_i, release_i], true)$: at most one of the state $critic_i$ and $release_i$ is equal to $true$ at the same time. ⌟

Example 8.4 (distinct)**.** The transition system in Figure 8.3 models the general problem of assigning a unique resource—here a unique identifier (UID)—to each process. The program graph on the left is assigning a unique value to the variable *uid*. The access to this variable is secured by the program graph on the right. The structural property to be checked is the uniqueness of the UID for the N concurrent instances of the left program. To this end, the global constraint `distinct(array)` ensures that the variables in *array* are all different. Therefore, the property is expressed with the constraint `distinct(`$\forall i \in N.uid_i$`)`. ⌟

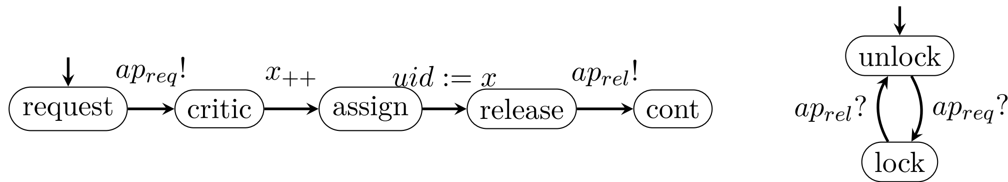


Figure 8.3: A transition system modeling the assignment of a UID to each process.

8.4.2 Constraint entailment algorithm

A logic formula is viewed as a spacetime process pruning the state space or stopping the exploration. It works by composing the process formula with the `StateSpace` module.

Given a constraint c, we propose a search strategy to establish the entailment of:

$$Sol_{3to2}(\langle d', P' \rangle) \models_{\mathscr{A}(Sol_2)} Sol_{3to2}(\langle d, P \rangle)$$

that was discussed in Section 2.4.2 with the lattice-based semantics of CCP. This algorithm works concurrently to the algorithm establishing the consistency of a CSP, and thus the CSP is not solved multiple times. However, we do not pause up the constraint search anymore when reaching the first solution, but on a criterion depending on the formula to check. The following class `Entailment` mimics the mathematical definition of the entailment and stops whenever a contradiction is encountered (the property can be $true$ or $false$):

```
class Entailment {
  world_line VStore domains;
  world_line CStore constraints;
  world_line StackLR queue;

  world_line L<Boolean> entailed = bot;
  single_space CStore property;

  public Entailment(VStore domains, CStore constraints
    CStore property, StackLR queue) { ... }

  public proc verify =
    universe(world_line domains, world_line constraints,
          single_space property, single_space entailed) with queue in
      par
      || entailment_on_solution()
      || pause_up_on_contradiction()
      end
    end

  flow entailment_on_solution =
    single_time ES consistent = read constraints.consistent(read domains);
```

```
      when consistent == true then
        entailed <- read property.consistent(read domains);
      end

    proc pause_up_on_contradiction =
      loop
        when entailed == top then
          pause up;
        else
          pause
        end
      end
  }
```

The field `entailed` reflects the status of the variable `property` in the current CSP. This variable is used to specify how the property is checked, for example to select in which state it must apply. We check the entailment of the property in each solution node, and we stop the algorithm whenever the property is both entailed and disentailed (`entailed` equals to the top element of its lattice).

For example, the property *at_most* presented in the Example 8.3 must be valid in every state of the transition system. This can be checked with the following algorithm:

```
  class ModelChecking {
    world_line VStore domains;
    world_line CStore constraints;
    world_line TransitionSystem ts;
    single_space CStore property;

    world_line StackLR queue = new StackLR();

    public ModelChecking(VStore domains, CStore constraints
      TransitionSystem ts, CStore property) { ... }

    proc check =
      single_space StateSpace state_space =
        new StateSpace(domains, constraints, ts, queue);
      single_space Entailment entailment =
        new Entailment(domains, constraints, property, queue);
      weak abort when entailment.entailed == false in
        par
        || state_space.search()
        || entailment.verify()
        end
      end
  }
```

Whenever we detect that the entailment is not consistent, we abort the computation. Thanks to the semantics of spacetime, the universes of both the CSP

solving and the property checking are combined. In addition, we use a nested combinatorial structure where the top-level universe is nondeterministic over a set of transitions, and where the lower universe is nondeterministic over a CSP.

The advantage of spacetime in this example is that we can easily add new strategies without modifying the existing code.

8.5 Conclusion

In this chapter, we sketched a framework combining constraint solving and model checking. The advantage of combining both techniques is that we do not need to restart the model checker for every different input, but instead we rely on a CSP to represent the domains of the input. The main purpose of this chapter was to show that spacetime can be applied outside the sole field of constraint programming. Moreover, it demonstrated that nested combinatorial structures can be expressed with universes, and that the resulting program is modular. This work is only a first step towards a usable model checker with constraints, and we need to assess the efficiency of such a system in the future. To conclude this chapter, we discuss about related work combining CSP and model checking.

Related work

Combining constraint solving and model checker problems is not a new concept as attested by the numerous works in the literature. There are numerous works on encoding model checking problems into a CSP to check safety and soundness properties [DP01, DP99, Pod00]. In particular, [NL00] encodes the CTL semantics using constraints clauses and an array reduction method to handle state space explosions. However, the problem (model transitions and formula) is often encoded in a huge constraint system, which does not merge the dynamic aspect of model checking, and the partial information management of constraint solving. Moreover, these works are mainly applied to specific applications, and the solution is generally to translate a model checking problem into a CSP.

In [LLLS11], the CSP is extended on infinite variable streams using Büchi-automata, a standard model checking technique, to solve the problem. They defined a stream CSP (called St-CSP) which provides streams to store the history of variables. The original aspect of this work is to use constraint and model checking to generate a stream of solutions verifying some properties. Therefore, they do not try to verify some properties on a model, but rather use these properties to generate valid solutions. They show it can be used to program applications with infinite behavior such as simulating juggling patterns and a musical harmonization problem. Interestingly, they are using a dataflow synchronous language as a specification language, and in this respect it would be very interesting to investigate the relation with spacetime.

Part IV

Conclusion

Chapter 9

Conclusion

The main goal of this dissertation was to unravel compositionality issues of search strategies in combinatorial problems. We first devised a lattice theory of constraint programming that precisely captures the complex combinatorial structures manipulated during constraint solving. It is given in a hierarchy of lattice structures isolating various components of a constraint solver. Beyond these algorithmic aspects, we also showed the lattice hierarchy was suited for classifying constraint-based languages. Our work is definitely not exhaustive and many constraint algorithms and languages were left out of scope. However, we hope that this new viewpoint reassembles the disparate pieces of constraint solving into a single theory.

To summarize our proposal, spacetime programming is centered around the concept of logical time as a synchronization mechanism of search strategies. Our idea is to view search strategies as processes. Hence, we can combine strategies with the very same operators as the ones used to combine processes. In general, a major challenge when designing a new process calculus is to ensure that the interactions between processes lead to a deterministic computation. In synchronous languages such as Esterel, they provide deterministic guarantees with the limitation that processes can only communicate through boolean variables. We generalized this model to arbitrary lattice structures which enable processes to communicate over structures as complex as constraints. Our causality analysis verifies that programs are monotonic functions, and thus preserve determinism.

This first extension allowed us to define processes communicating inside one layer of the lattice hierarchy. We further extended the language with spacetime refinement in order to program processes operating over multiple layers of the lattice hierarchy. It generalizes time refinement, a notion of nested time-scale already presents in synchronous languages such as Quartz. In addition to the time dimension, we propose a spatial dimension with an abstract queueing strategy that links the layers of the hierarchy. Spacetime refinement opens the door to many more search strategies such as the family of restart-based search strategies.

After formalizing the semantics of spacetime, we turned to the applications of our language. Our musical interactive computer-aided composition software

demonstrated that spacetime is suited to program search strategies that can be easily stopped and resumed. One of the advantage is to interact with the graphical user interface during search such that the user can impact the search process. We focussed on a first proof of concept and we did not consider the ergonomic aspect and the user experience of this software. In a second time, we applied spacetime programming to model checkers where we merged ideas from constraint programming and model checking algorithms. The departure point of this work was the observation that parametric model checking could be solved more efficiently with constraint solving. We wrote a search strategy that dynamically creates a CSP as we explore the model. Although these works are both experimental, it gives a glimpse into the applicability of spacetime programming beyond the field of constraint programming.

The concept of logical time inside the semantics of programming languages has been very little explored outside the field of synchronous programming. We hope this dissertation demonstrates that time is a central device to coordinate and compose processes. In addition to time, we organized the space as a lattice structure, and we showed that usual problems of deadlock and race conditions vanished with compile-time analysis. Time and space are central concepts in physics but also in all sciences and in nature, and we believe they are essential concepts in programming languages as well.

Chapter 10

Future work

10.1 Towards constraint programming with lattices

Our lattice framework developed in Part I is a first attempt to characterize constraint programming with lattices. We believe that it can be extend to better reflect the techniques employed in constraint solvers. First, we sketch how to use the delta operation to program restoration strategies, which are crucial to manage memory in constraint solvers. Secondly, we discuss the equivalence of two CSPs which is an effective technique to simplify constraint models.

Lattice for restoration strategies

A transversal concern when backtracking is the restoration strategy used for storing nodes in a compact way; this is crucial to solve large CSP's instances in constraint solvers. In the lattice L_4, we build a search tree by fully copying the nodes of the search tree. However, it has the disadvantage of consuming a lot of memory and solvers are usually based on other restoration strategies. We can mention three main restoration strategies: copying, trailing and recomputation [Sch99, CHN01, Sch02]. One of the only competitive systems using copying is Minion [GJM06]. It is competitive because it targets a specific class of problems and optimizes the memory at the bit-level. Generally, constraint systems either use trailing (e.g. Choco [PFL15] and Eclipse [SS12]) or recomputation (e.g. Oz [Sch02] and GeCode [STL14]). Trailing is inherited from the constraint logic paradigm (see Section 2.5.2) while recomputation is more recent but has been proven very competitive [Sch99].

The delta operator Δ forms the basis for integrating such restoration strategies into our lattice framework. We provide an example supporting this claim.

Example 10.1 (Difference between two nodes)**.** We first recall the CSP *NeqXYZ* presented in Example 1.8:

$$\begin{aligned} NeqXYZ = \langle\ & \{x \mapsto [1..1], y \mapsto [1..3], z \mapsto [1..3]\}, \\ & \{[\![x \neq y]\!], [\![x \neq z]\!], [\![y \neq z]\!]\}\rangle \end{aligned}$$

We consider the delta between the top element of $TS(NeqXYZ)$ given by

$$T = \langle \quad \{x \mapsto [1..1], y \mapsto [2..3], z \mapsto [2..3]\}, \\ \{[\![x \neq y]\!], [\![x \neq z]\!], [\![y \neq z]\!]\}\rangle$$

and the left node obtained after increasing the information in y given by

$$L = \langle \quad \{x \mapsto [1..1], y \mapsto [2..2], z \mapsto [2..3]\}, \\ \{[\![x \neq y]\!], [\![x \neq z]\!], [\![y \neq z]\!]\}\rangle$$

The right node R is defined similarly. The delta between T and L is given by $DL = T\Delta L = \langle\{y \mapsto [2..2])\}, \emptyset\rangle$ and between T and R by $DR = T\Delta R = \langle\{y \mapsto [3..3]\}, \emptyset\rangle$. The set $\{T, DL, DR\}$ is more compact than the full queue of nodes $\{L, R\}$. The key is to avoid repeating information present in different nodes. Nevertheless, the full queue can be recovered with $\{T \sqcup_3 DL, T \sqcup_3 DR\}$. Therefore, it is a lossless compression of the lattice L_4. ⌟

A research direction is to investigate the various forms of trailing with the operator Δ applied to different layers of the hierarchy, and recomputation as a delta over the constraint store in L_3. Moreover, local search algorithms also work with the delta operation to compute the distance between two states [VHM05]. Perhaps the delta operator can lead towards a language for expressing both local and global search with similar language's abstractions.

Equivalence of two CSPs

In the lattice L_3, we order the propagator set $\mathcal{P}(Prop)$ by set inclusion. This order seems a bit restrictive since, for example, the two sets of constraints $\{x > y\}$ and $\{x \neq y, x \geq y\}$ are unordered. However, these two sets are equivalent since they generate the same set of solutions. Applications of verifying the equivalence occur, for example, in the modelling language MiniZinc where a preprocessing step is applied to the models when they are compiled to the simpler format FlatZinc [LT15]. This is useful to reduce the solving time and to keep low the number of concepts to be implemented by the back-end solvers. At the theoretical level, such preprocessing algorithms have roots into theorem provers since $CSP_1 \iff CSP_2$ must be a valid formula. From this perspective, constraint programming in the context of theorem-proving has been studied in [Apt98, Apt03]. The question is how to incorporate such equivalences in our lattice framework, and is it relevant to do so?

10.2 Extensions of spacetime

In this section, we consider three extensions of spacetime. Firstly, spacetime can be extended to provide a better support for user-defined lattice structures. It would allow the user to create more easily its own lattices, and enable the compiler to perform better static analysis. Secondly, we consider higher-order processes

in spacetime, which improve the capability of abstracting strategies with similar search patterns. In a third extension, we investigate a stronger causality analysis by comparison with what is realized in Esterel.

Lattice type

The semantics of spacetime assumes that the variables and host functions are well-defined. For example, the user must provide the implementation, in the host language, of the entailment and join operations for each variable's type. It can be tricky and error-prone, especially for users not acquainted with the lattice theory. Therefore, it is pertinent to investigate *lattice types* for defining lattice structures within spacetime.

We saw in Chapter 1 that most lattices are defined by composition of existing lattices. The idea is to provide a collection of basic lattices. From these, the user can derive new well-formed lattices using composition operators inside spacetime (e.g. Cartesian product, disjoint union; see Section 1.2). It is interesting to realize that compound data (structures or objects) are analogue to Cartesian products, and that sum types are similar to disjoint unions. Perhaps type theory can readily formalize lattice composition.

Lattice types raise several questions that must be answered in future research. Is it possible to provide a complete (or sufficiently interesting) set of declarative operators for composing lattices? How to define the entailment and join operations of the newly created lattices with spacetime operations?

Higher-order spacetime process

The search strategies `IDS` and `LDS` introduced in Chapter 6 follow a very similar pattern: we explore a tree until a condition is met, and we restart by increasing a bound. However, we duplicated this behavior for both strategies instead of isolating it into a higher-order strategy. It is interesting to take an example from the search combinators [STW$^+$13] that is specifically designed to specify such strategies:

$$
\begin{aligned}
&id(s) \stackrel{\text{def}}{=} ir(depth, 0, +, 1, \infty, s) \\
&ir(p, l, \oplus, i, u, s) \stackrel{\text{def}}{=} let(n, l, restart(n \leq u, \\
&\qquad and([assign(n, n \oplus i), limit(p \leq n, s)])))
\end{aligned}
$$

The combinator id is an iterative depth-first search (IDS) that restarts a strategy s following the pattern of IDS. They encapsulate iterative restarting strategy in a combinator ir where the strategy s is restarted until we reach a limit $n \leq u$. To summarize, n is an internal counter initialized at l, and increased by $n \oplus i$ on each restart.

What is important is that the search strategy is written *vertically*: each strategy is encapsulated in another strategy. In spacetime, we compose search strategy

horizontally: each strategy is executed concurrently ("next to") another strategy. We believe that both vertical and horizontal compositionality is required in order to achieve high re-usability of search strategies.

However, the causality analysis requires that the full program is known at compile-time, and thus we need to fully expand the code to perform this analysis. Our proposal is to rely on the capabilities of the host language to obtain such higher-order processes. In the following example, we propose a restart-based search strategy parametrized by a bounded search process such as `BoundedDepth` and `BoundedDiscrepancy` (see Section 6.1.2).

```
interface BoundedSearch {
  public proc prune_on_limit(single_space LMax);
}

class RestartSearch<Queue, T extends BoundedSearch> {
  single_space LMax limit = new LMax(0);
  single_time Queue queue;
  single_time T bound = new T();

  public RestartSearch(Queue queue) { ... }

  flow search =
    readwrite limit.inc();
    universe(single_space limit, single_space bound) with queue in
      bound.prune_on_limit(limit);
    end
}
```

The class `RestartSearch` is parametrized by the type of a queue and by a bounded search strategy. We expect this strategy to implement the interface `BoundedSearch` in order to call its process `prune_on_limit`. We can instantiate this class with a `BoundedDepth` type as follows:

```
single_time StackLR queue = new StackLR();
RestartSearch<StackLR, BoundedDepth> ids = new RestartSearch<>(queue);
ids.search();
```

However, such an extension is not straightforward to integrate with the host language `Java` due to the type-erasure scheme in Java. To keep the code known at compile-time, spacetime would need an expansion mechanism similar to the `C++` templates. A possibility is to expand the code at compile-time for the analysis, and to rely on the type-erasure scheme of `Java` when generating the code. Besides, another problem occurs: we need to take care of the other forms of polymorphism. For example, overriding is a dynamic mechanism in `Java`, and thus it does not mix well with static analysis.

Stronger causality analysis of spacetime program

To perform its causality analysis, Esterel performs a partial evaluation of the program by considering the value of the signals. It is interesting to pursue this idea with spacetime programs in order to accept more causal programs. However, this can only be achieved by a deep integration into the host language since the compiler of spacetime would need to cooperate with the host lattice structures. Another possibility, converging with the idea of lattice modules, is to define the lattice structures within spacetime. It would enable the compiler to reason more precisely about these lattices.

10.3 Beyond search in constraint programming

In this section, we investigate how spacetime could be integrated with other paradigms in order to program the layer L_3 in the lattice hierarchy.

Programming L_3 in spacetime

Guarded commands are a first step towards programming in spacetime the lattice L_3, or more precisely, the transition system sub-lattice TS_3. One of the challenges is to be able to customize what branch is selected next within the spacetime language. This would allow implementers to program various propagation engines with different selection strategies such as with the DSL presented in [PLDJ14]. Similarly to search strategies, the inference strategy can have a substantial impact on the performances.

Another challenge is to program global constraints which are usually implemented in lower-level programming languages. A research direction would be to use a graph rewriting language, such as LMNtal [Ued09], for programming global constraints, and to obtain a fully general language for programming the lattice L_3. It is especially important because global constraints are at the heart of constraint programming, and entire researches are axed on specific global constraints. On the other hand, global constraints are also central to bridging constraint programming with other fields such as data mining [GDN$^+$15].

To program L_3 in spacetime, an idea is to generalize the fixed point semantics of the entailment operator to arbitrary functions. For example, we could propose an operator `fixpoint fun(a, b)` that would call the host function fun several times until it reaches a fixed point and whenever a and b are modified. Such a fixed point operator would allow the propagators to be programmed and to interact under some events. A challenge is to extend the deterministic scheduling of read-write accesses to deal with fixed point functions.

Bibliography

[AADR98] Carlos Agon, Gérard Assayag, Olivier Delerue, and Camilo Rueda. Objects, time and constraints in OpenMusic. In *Proceedings of the 1998 International Computer Music Conference*, 1998.

[ABPS98] Krzysztof R. Apt, Jacob Brunekreef, Vincent Partington, and Andrea Schaerf. Alma-O: An Imperative Language That Supports Declarative Programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(5):1014–1066, September 1998.

[AGM15] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. A Multicore Tool for Constraint Solving. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI '15, pages 232–238, 2015.

[AH10] Sergio Antoy and Michael Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74, 2010.

[AKP93] Hassan Ait-Kaci and Andreas Podelski. Towards a meaning of LIFE. *The Journal of Logic Programming*, 16(3-4):195–234, 1993.

[AM11] Torsten Anders and Eduardo R. Miranda. Constraint programming systems for modeling music theories and composition. *ACM Comput. Surv.*, 43(4):30:1–30:38, October 2011.

[Apt98] Krzysztof R. Apt. A proof theoretic view of constraint programming. *Fundam. Inf.*, 34(3):295–321, August 1998.

[Apt03] Krzysztof Apt. *Principles of constraint programming.* Cambridge University Press, 2003.

[AS99] Krzysztof R. Apt and Andrea Schaerf. *The Alma Project, or How First-Order Logic Can Help us in Imperative Programming*, pages 89–113. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[AW07] Krzysztof R Apt and M Wallace. *Constraint logic programming using ECLiPSe.* Cambridge University Press, 2007. OCLC: 181382157.

[Bah93] Reem Bahgat. *Non-Deterministic Concurrent Logic Programming in Pandora.* World Scientific Publishing Company, 1993.

[Bar03] Chitta Baral. *Knowledge representation, reasoning, and declarative problem solving.* Cambridge University Press, Cambridge; New York, 2003. OCLC: 56416111.

[BCFP13] Nicolas Beldiceanu, Mats Carlsson, Pierre Flener, and Justin Pearson. On the reification of global constraints. *Constraints*, 18(1):1–6, January 2013.

[BCLGH93] Albert Benveniste, Paul Caspi, Paul Le Guernic, and Nicolas Halbwachs. Data-flow synchronous languages. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, pages 1–45. Springer, 1993.

[BCR11] Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global Constraint Catalog, 2nd Edition (revision a), February 2011. SICS research report T2012-03, http://soda.swedish-ict.se/5195/.

[Ber00a] Gérard Berry. *The Esterel v5 language primer: version v5_91.* Centre de mathématiques appliquées, Ecole des mines and INRIA, 2000.

[Ber00b] Gérard Berry. The Foundations of Esterel. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction*, pages 425–454. MIT Press, 2000.

[Ber02] Gerard Berry. The constructive semantics of pure Esterel. draft version 3. 2002.

[BG92] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87 – 152, 1992.

[BG12] Sébastien Bardin and Arnaud Gotlieb. FDCC: a combined approach for solving constraints over finite domains and arrays. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 17–33. Springer, 2012.

[BHvMW09] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications.* IOS Press, February 2009.

[BJMP13] Guillaume Baudart, Florent Jacquemard, Louis Mandel, and Marc Pouzet. A synchronous embedding of antescofo, a domain-specific language for interactive mixed music. In *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 1–12. IEEE, 2013.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, May 2008.

[BPP95] Frank S. de Boer, Alessandra Di Pierro, and Catuscia Palamidessi. *Nondeterminism and Infinite Computations in Constraint Programming*. 1995.

[Bra12] Ivan Bratko. *Prolog programming for artificial intelligence*. Pearson education, 4th edition, 2012.

[BS00] Frédéric Boussinot and Jean-Ferdy Susini. Java threads and SugarCubes. *Software: Practice and Experience*, 30(5):545–566, 2000.

[BS14] Gérard Berry and Manuel Serrano. Hop and HipHop: Multitier Web Orchestration. In *International Conference on Distributed Computing and Internet Technology*, pages 1–13. Springer, 2014.

[BTM11] Anicet Bart, Charlotte Truchet, and Eric Monfroy. Verifying a real-time language with constraints. pages 844–851. IEEE, 2015-11.

[BV18] Paolo Boldi and Sebastiano Vigna. On the Lattice of Antichains of Finite Intervals. *Order*, 35(1):57–81, March 2018.

[CB00] Carlos Castro and Peter Borovanskỳ. The use of a strategy language for solving search problems. *Annals of Mathematics and Artificial Intelligence*, 29(1):35–64, 2000.

[CCD94] Björn Carlson, Mats Carlsson, and Daniel Diaz. Entailment of finite domain constraints. *ICLP'94, International Conference on Logic Programming*, 1994.

[CD96] Philippe Codognet and Daniel Diaz. Compiling constraints in clp(fd). *The Journal of Logic Programming*, 27(3):185 – 226, 1996.

[CG81] Keith L. Clark and Steve Gregory. A relational language for parallel programming. In *Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pages 171–178. ACM, 1981.

[CG83] Keith L Clark and Steve Gregory. PARLOG: A parallel logic programming language. *Research Report DOC*, 83(5), 1983.

[CGK+13] Sylvain Conchon, Amit Goel, Sava Krstic, Alain Mebsout, and Fatiha Zaïdi. Invariants for finite instances and beyond. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 61–68, 2013.

[CHN01] Chiu Wo Choi, Martin Henz, and Ka Boon Ng. Components for state restoration in tree search. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, CP '01, pages 240–255, London, UK, UK, 2001. Springer-Verlag.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[CL01] JaSon Crampton and George Loizou. The completion of a poset in a lattice of antichains. *International Mathematical Journal*, 1(3):223–238, 2001.

[Cla78] KeithL. Clark. Negation as failure. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Springer US, 1978.

[CM12] Mats Carlsson and Per Mildner. SICStus Prolog - the first 25 years. *Theory and Practice of Logic Programming*, 12(1):35–66, 2012.

[CMA+12] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 1. ACM, 2012.

[Col85] Alain Colmerauer. Prolog in 10 figures. *Communications of the ACM*, 28(12):1296–1310, 1985.

[Col90] Alain Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1990.

[Col93] Alain Colmerauer. The birth of prolog. In *III, CACM Vol.33, No7*, pages 37–52, 1993.

[CRVH08] Hélene Collavizza, Michel Rueher, and Pascal Van Hentenryck. CPBPV: A constraint-programming framework for bounded program verification. In *International Conference on Principles and Practice of Constraint Programming*, pages 327–341. Springer, 2008.

[CWY91] Vitor Santos Costa, David HD Warren, and Rong Yang. *Andorra I: a parallel Prolog system that transparently exploits both And-and or-parallelism*, volume 26. ACM, 1991.

[DAC12] Daniel Diaz, Salvador Abreu, and Philippe Codognet. On the implementation of GNU prolog. *Theory and Practice of Logic Programming*, 12(1):253–282, 2012.

[dBP94] Frank S. de Boer and Catuscia Palamidessi. From concurrent logic programming to concurrent constraint programming. *Advances in Logic Programming Theory, Oxford University Press*, 1994.

[DC93] Daniel Diaz and Philippe Codognet. A Minimal Extension of the WAM for clp (FD). In *Proceedings of the Tenth International Conference on Logic Programming*, pages 774–790, 1993.

[DC01] Daniel Diaz and Philippe Codognet. Design and Implementation of the GNU Prolog System. *Journal of Functional and Logic Programming*, 6(2001):542, 2001.

[DdlBS04] Gregory J. Duck, María García de la Banda, and Peter J. Stuckey. Compiling ask constraints. In Bart Demoen and Vladimir Lifschitz, editors, *Logic Programming*, volume 3132, pages 105–119. Springer Berlin Heidelberg, 2004. DOI: 10.1007/978-3-540-27775-0_8.

[DF02] Rina Dechter and Daniel Frost. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136(2):147 – 188, 2002.

[Dij75] Edsger W. Dijkstra. Guarded commands, non-determinacy and a calculus for the derivation of programs. *SIGPLAN Not.*, 10(6):2–2.13, April 1975.

[DJ12] Gregory J. Duck and Joxan Jaffar. CLP entailment as lazy clause generation. 2012.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[DP99] Giorgio Delzanno and Andreas Podelski. *Model Checking in CLP*, pages 223–239. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[DP01] Giorgio Delzanno and Andreas Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, 2001.

[DP02] B. A. Davey and H. A. Priestley. *Introduction to lattices and order*. Cambridge University Press, 2002.

[ESC16] Rémy El Sibaïe and Emmanuel Chailloux. Synchronous-reactive web programming. pages 9–16. ACM Press, 2016.

[FBB92] Bjorn N. Freeman-Benson and Alan Borning. The design and implementation of kaleidoscope'90-a constraint imperative programming language. In *Computer Languages, 1992., Proceedings of the 1992 International Conference on*, pages 174–180. IEEE, 1992.

[FBH13] Tim Felgentreff, Alan Borning, and Robert Hirschfeld. *Babelsberg: specifying and solving constraints on object behavior*. Number 81 in Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam. Univ.-Verl, Potsdam, 2013. OCLC: 931548168.

[FGK90] Robert Fourer, David M. Gay, and Brian W. Kernighan. A Modeling Language for Mathematical Programming. *Management Science*, 36(5):519–554, May 1990.

[FKF+93] Kazuhiro Fuchi, Robert Kowalski, Koichi Furukawa, Kazunori Ueda, Ken Kahn, Takashi Chikayama, and Evan Tick. Launching the new era. *Commun. ACM*, 36(3):49–100, March 1993.

[FMBH15] Tim Felgentreff, Todd Millstein, Alan Borning, and Robert Hirschfeld. Checks and balances: constraint solving without surprises in object-constraint programming languages. In *ACM SIGPLAN Notices*, volume 50, pages 767–782. ACM, 2015.

[Frü98] Thom Frühwirth. Theory and practice of constraint handling rules. *The Journal of Logic Programming*, 37(13):95 – 138, 1998.

[FS09] Alan M. Frisch and Peter J. Stuckey. The Proper Treatment of Undefinedness in Constraint Languages. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, CP '09, pages 367–382, Berlin, Heidelberg, 2009. Springer-Verlag.

[GBS13] Mike Gemünde, Jens Brandt, and Klaus Schneider. Clock refinement in imperative synchronous languages. *EURASIP Journal on Embedded Systems*, 2013(1):1–21, 2013.

[GDN+15] Tias Guns, Anton Dries, Siegfried Nijssen, Guido Tack, and Luc De Raedt. MiningZinc: A declarative framework for constraint-based mining. *Artificial Intelligence*, 2015.

[GG93] Ralph E. Griswold and Madge T. Griswold. History of the Icon Programming Language. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, pages 53–68, New York, NY, USA, 1993. ACM.

[GG97] Ralph E. Griswold and Madge T. Griswold. *The Icon programming language*. Peer-to-Peer Communications, San Jose, Calif., U.S.A, 3rd edition, 1997.

[GH04] Martin Grabmüller and Petra Hofstedt. Turtle: A Constraint Imperative Programming Language. In Frans Coenen, Alun Preece, and Ann Macintosh, editors, *Research and Development in Intelligent Systems XX: Proceedings of AI2003, the Twenty-third SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence*, pages 185–198. Springer London, London, 2004. DOI: 10.1007/978-0-85729-412-8_14.

[GJM06] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *ECAI*, volume 141, pages 98–102, 2006.

[Göd30] Kurt Gödel. Die vollstandigkeit der axiome des logischen funktionenkalkiils. *Monatshefte für mathematik und physik*, 37:349–360, 1930.

[Göd31] Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für mathematik und physik*, 38(1):173–198, 1931.

[GPA+01] Gopal Gupta, Enrico Pontelli, Khayri AM Ali, Mats Carlsson, and Manuel V. Hermenegildo. Parallel execution of prolog programs: a survey. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(4):472–602, 2001.

[GPR96] Roberto Giacobazzi, Catuscia Palamidessi, and Francesco Ranzato. Weak relative pseudo-complements of closure operators. *Algebra Universalis*, 36(3):405–412, 1996.

[Gra03] Martin Grabmüller. The Constraint Imperative Programming Language Turtle. In *20. Workshop der GI-Fachgruppe*, volume 2, 2003.

[Gre69] Cordell Green. Application of theorem proving to problem solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, IJCAI ’69, pages 219–239, San Francisco, CA, USA, 1969. Morgan Kaufmann Publishers Inc.

[GW99] Ian P. Gent and Toby Walsh. *CSPlib: A Benchmark Library for Constraints*, pages 480–481. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[HA28] David Hilbert and Wilhelm Ackermann. *Grundzüge der theoretischen Logik*. 1928.

[Hal92] Nicolas Halbwachs. *Synchronous programming of reactive systems*. Number 215. Springer Science & Business Media, 1992.

[Han07] Michael Hanus. Multi-paradigm declarative languages. In *ICLP*, volume 7, pages 45–75. Springer, 2007.

[Han14] Michael Hanus. Multiparadigm languages. In *Computing Handbook, Third Edition: Computer Science and Software Engineering*, pages 66: 1–17. 2014.

[Har90] Seif Haridi. A logic programming language based on the Andorra model. *New Generation Computing*, 7(2-3):109–125, 1990.

[HD86] P. Van Hentenryck and M. Dincbas. Domains in Logic Programming. In *Proceedings of the Fifth AAAI National Conference on Artificial Intelligence*, AAAI '86, pages 759–765, Philadelphia, Pennsylvania, 1986. AAAI Press.

[Her30] Jacques Herbrand. *Recherches sur la théorie de la démonstration*. PhD thesis, University of Paris, Paris, 1930.

[Hew69] Carl Hewitt. PLANNER: A language for proving theorems in robots. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, IJCAI '69, pages 295–301, San Francisco, CA, USA, 1969. Morgan Kaufmann Publishers Inc.

[Hew71] Carl Hewitt. Procedural embedding of knowledge in planner. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, IJCAI '71, pages 167–182, San Francisco, CA, USA, 1971. Morgan Kaufmann Publishers Inc.

[Hew09] Carl Hewitt. Inconsistency robustness in logic programs. *arXiv preprint arXiv:0904.3036*, 2009.

[HF11] Gerard J. Holzmann and Mihai Florian. Model checking with bounded context switching. *Formal Aspects of Computing*, 23(3):365–389, 2011.

[HG95] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceedings of the 14th International Conference on Artificial Intelligence*, IJCAI '95, pages 607–615, 1995.

[HJ90] Seif Haridi and Sverker Janson. Kernel Andorra Prolog and its computation model. In *Logic Programming: Proceedings of the Seventh International Conference*, Jerusalem, Israel, June 1990. MIT Press.

[HK06] Petra Hofstedt and Olaf Krzikalla. *TURTLE++ – A CIP-Library for C++*, pages 12–24. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[HK12] Jacob M. Howe and Andy King. A pearl on SAT and SMT solving in prolog. *Theoretical Computer Science*, 435:43–55, June 2012.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, August 1978.

[Hof11] Petra Hofstedt. *Multiparadigm Constraint Programming Languages.* Cognitive Technologies. Springer Berlin Heidelberg, 2011.

[Hol92] Christian Holzbaur. Metastructures vs. attributed variables in the context of extensible unification. In *Programming Language Implementation and Logic Programming*, pages 260–268. Springer, 1992.

[Hoo02] John N. Hooker. Logic, optimization, and constraint programming. *INFORMS Journal on Computing*, 14(4):295–321, 2002.

[Hoo12] John N. Hooker. *Integrated Methods for Optimization*, volume 170 of *International Series in Operations Research & Management Science.* Springer US, Boston, MA, 2012. DOI: 10.1007/978-1-4614-1900-6.

[HS98] M. Hanus and F. Steiner. Controlling Search in Declarative Programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pages 374–390. Springer LNCS 1490, 1998.

[IBM15] IBM. *IBM ILOG CPLEX Optimization Studio CP Optimizer Users Manual—Version 12 Release 6.* 2015.

[JH91] Sverker Janson and Seif Haridi. Programming Paradigms of the Andorra Kernel Language. In *Logic Programming: Proceedings of the 1991 International Logic Programming Symposium*, San Diego, California, October 1991. MIT Press. (Revised version of SICS Research Report R91:08).

[JL87] Joxan Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119. ACM, 1987.

[JM94] Joxan Jaffar and Michael J. Maher. Special issue: Ten years of logic programming constraint logic programming: a survey. *The Journal of Logic Programming*, 19:503 – 581, 1994.

[JMSY92] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland HC Yap. The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(3):339–395, 1992.

[JUIS94] Sverker Janson, Uppsala universitet, Institutionen för ADB och datalogi, and Swedish Institute of Computer Science. AKL, a multi-paradigm programming language: based on a concurrent constraint framework, 1994.

[Kor85] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[Kor96] Richard E. Korf. Improved limited discrepancy search. In *In Proceedings of AAAI-96*, pages 286–291. MIT Press, 1996.

[Kow74] R. A. Kowalski. Predicate logic as a programming language. In *Proc IFIP Cong*, pages 569–574. North-Holland Pub Co, 1974.

[Kow79] Robert Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22(7):424–436, 1979.

[Kow14] Robert Kowalski. Logic programming. *Handbook of the History of Logic*, 9:523 – 569, 2014. Computational Logic.

[KPH04] Josef Kallrath, Panos M. Pardalos, and Donald W. Hearn, editors. *Modeling Languages in Mathematical Optimization*, volume 88 of *Applied Optimization*. Springer US, Boston, MA, 2004. DOI: 10.1007/978-1-4613-0215-5.

[KS08] Daniel Kroening and Ofer Strichman. *Decision procedures: an algorithmic point of view*. Texts in theoretical computer science. Springer, 2008. OCLC: 244022161.

[Lau78] Jena-Lonis Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1):29 – 127, 1978.

[Lau96a] Jean-Louis Laurière. Programmation de contraintes ou programmation automatique? Technical report, Université Pierre et Marie Curie, 1996.

[Lau96b] Mikael Laurson. *PatchWork: A visual programming language and some musical applications*. PhD thesis, 1996.

[LC98] François Laburthe and Yves Caseau. Salsa: A Language for Search Algorithms. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Michael Maher, and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming CP98*, volume 1520, pages 310–324. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. DOI: 10.1007/3-540-49481-2_23.

[Lee06] Edward A. Lee. The problem with threads. Technical report, EECS Department, University of California, Berkeley, 2006.

[LFBB94] Gus Lopez, Bjorn Freeman-Benson, and Alan Borning. Kaleidoscope: A Constraint Imperative Programming Language. In Brian Mayoh, Enn Tyugu, and Jaan Penjam, editors, *Constraint Programming*, pages 313–329. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994. DOI: 10.1007/978-3-642-85983-0_12.

[LL14] Jasper CH Lee and Jimmy HM Lee. Towards practical infinite stream constraint programming: Applications and implementation. In *International Conference on Principles and Practice of Constraint Programming*, pages 449–464. Springer, 2014.

[LLLS11] Arnaud Lallouet, Yat Chiu Law, Jimmy HM Lee, and Charles FK Siu. Constraint programming on infinite data streams. In *International Joint Conference on Artificial Intelligence*, pages 597–604, 2011.

[LT15] Kevin Leo and Guido Tack. Multi-Pass High-Level Presolving. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI '15, pages 346–352, 2015.

[Mah87] Michael J. Maher. Logic Semantics for a Class of Committed-choice Programs. *Proceedings of the Fourth International Conference on Logic Programming*, pages 858–876, 1987.

[Man06] Louis Mandel. Conception, sémantique et implantation de ReactiveML : un langage à la ML pour la programmation réactive, 2006.

[MB05] Louis Mandel and Farid Benbadis. Simulation of mobile ad hoc network protocols in ReactiveML. In *Proceedings of Synchronous Languages, Applications, and Programming (SLAP 2005)*, Edinburgh, Scotland, 2005. Electronic Notes in Theoretical Computer Science.

[Mes97] Pedro Meseguer. Interleaved depth-first search. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, volume 2 of *IJCAI '97*, pages 1382–1387, 1997.

[MFS15] Thierry Martinez, François Fages, and Sylvain Soliman. Search by constraint propagation. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, PPDP '15, pages 173–183. ACM Press, 2015.

[MJMS98] K. Marriott, J. Jaffar, M. J. Maher, and P. J. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1):46, 1998.

[MPP15] Louis Mandel, Cédric Pasteur, and Marc Pouzet. Time refinement in a functional synchronous language. *Science of Computer Programming*, 111:190–211, 2015.

[MS74] Robert Morris and Daniel Starr. The structure of all-interval series. *Journal of Music Theory*, 18(2):364–389, 1974.

[MVH99] Laurent Michel and Pascal Van Hentenryck. Localizer: A modeling language for local search. *INFORMS Journal on Computing*, 11(1):1–14, 1999.

[NL00] Ulf Nilsson and Johan Lübcke. Constraint logic programming for local and symbolic model-checking. In *Proceedings of the First International Conference on Computational Logic*, CL '00, pages 384–398, London, UK, UK, 2000. Springer-Verlag.

[NM95] Ulf Nilsson and Jan Małuszyński. *Logic, Programming and Prolog.* 2ed edition, 1995.

[NSB+07] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming—CP 2007*, pages 529–543. Springer, 2007.

[Old93] William Older. Programming in CLP(BNR). In *In Position Papers for the First Workshop on Principles and Practice of Constraint Programming*, pages 239–249, 1993.

[ORV13] Carlos Olarte, Camilo Rueda, and Frank D. Valencia. Models and emerging trends of concurrent constraint programming. *Constraints*, 18(4):535–578, 2013.

[OSC09] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, September 2009.

[Pas13] Cédric Pasteur. Raffinement temporel et exécution parallèle dans un langage synchrone fonctionnel, 2013.

[PBEB07] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. Springer Publishing Company, Incorporated, 1st edition, 2007.

[PFL15] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*, 2015.

[PLDJ14] Charles Prudhomme, Xavier Lorca, Rémi Douence, and Narendra Jussien. Propagation engine prototyping with a domain specific language. *Constraints*, 19(1):57–76, 2014.

[Pod00] Andreas Podelski. Model checking as constraint solving. In Jens Palsberg, editor, *Static Analysis: 7th International Symposium, SAS 2000, Santa Barbara, CA, USA, June 29 - July 1, 2000. Proceedings*, pages 22–37. Springer Berlin Heidelberg, 2000. DOI: 10.1007/978-3-540-45099-3_2.

[PQZ12] Gilles Pesant, Claude-Guy Quimper, and Alessandro Zanarini. Counting-based search: Branching heuristics for constraint satisfaction problems. *J. Artif. Intell. Res.(JAIR)*, 43:173–210, 2012.

[Pra60] Dag Prawitz. An improved proof procedure. *Theoria*, 26(2):102–139, 1960.

[PRS16] Anthony Palmieri, Jean-Charles Régin, and Pierre Schaus. Parallel Strategies Selection. In *International Conference on Principles and Practice of Constraint Programming*, pages 388–404. Springer, 2016.

[PZ90] Miller Puckette and David Zicarelli. *MAX, Development Package*. Ircam and Opcode Systems, 1990.

[QW04] Shaz Qadeer and Dinghao Wu. KISS: keep it simple and sequential. *Acm sigplan notices*, 39(6):14–24, 2004.

[RGST15] Andrea Rendl, Tias Guns, Peter J. Stuckey, and Guido Tack. MiniSearch: a solver-independent meta-search language for MiniZinc. In *Principles and Practice of Constraint Programming*, pages 376–392. Springer, 2015.

[RH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.

[Ric76] John R. Rice. The Algorithm Selection Problem. volume 15 of *Advances in Computers*, pages 65 – 118. Elsevier, 1976. DOI: 10.1016/S0065-2458(08)60520-3.

[RMS96] Peter Van Roy, Michael Mehl, and Ralf Scheidhauer. Integrating efficient records into concurrent constraint programming. In *Programming Languages: Implementations, Logics, and Programs, 8th International Symposium, PLILP'96, Aachen, Germany, September 24-27, 1996, Proceedings*, pages 438–453, 1996.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[Rob00] John Alan Robinson. Computational logic: Memories of the past and challenges for the future. In *Computational LogicCL 2000*, pages 1–24. Springer, 2000.

[RRM13] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Embarrassingly parallel search. In *International Conference on Principles and Practice of Constraint Programming*, pages 596–610. Springer, 2013.

[RvBW06] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., 2006.

[Sar93] Vijay A. Saraswat. *Concurrent constraint programming*. ACM Doctoral dissertation awards. MIT Press, 1993.

[SBF10] Peter J. Stuckey, Ralph Becket, and Julien Fischer. Philosophy of the MiniZinc challenge. *Constraints*, 15(3):307–316, July 2010.

[SC00] Vítor Santos Costa. Parallelism and Implementation Technology for Logic Programming Languages. In *Encyclopedia of Computer Science and Technology*, volume 42, pages 197–237. Marcel Dekker Inc, 2000.

[Sch99] Christian Schulte. Comparing trailing and copying for constraint programming. In *In Proceedings of the International Conference on Logic Programming*, pages 275–289. The MIT Press, 1999.

[Sch00] Christian Schulte. Parallel search made simple. In *University of Singapore*, pages 41–57, 2000.

[Sch02] Christian Schulte. *Programming Constraint Services: High-Level Programming of Standard and New Constraint Services*, volume 2302 of *Lecture Notes in Computer Science*. Springer, 2002.

[Sch09] Klaus Schneider. The synchronous programming language Quartz, 2009.

[SDTD14] Tom Schrijvers, Bart Demoen, Markus Triska, and Benoit Desouter. Tor: Modular search with hookable disjunction. *Sci. Comput. Program.*, 84:101–120, May 2014.

[Seb07] Roberto Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:141–224, 2007.

[SFS+14] Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. The MiniZinc challenge 20082013. *AI Magazine*, 35(2):55–60, 2014.

[SGJ14] Vijay Saraswat, Vineet Gupta, and Radha Jagadeesan. TCC, with history. In *Horizons of the Mind. A Tribute to Prakash Panangaden*, pages 458–475. Springer, 2014.

[Sha83] Ehud Shapiro. A subset of concurrent prolog and its interpreter. *ICOT Technical Report, TR-003*, 1983.

[Sha86] Ehud Shapiro. Review of 'foundations of logic programming (lloyd, j.w.)'. *Computing Reviews*, 1986.

[Sha87] E. Shapiro, editor. *Concurrent Prolog: Collected Papers.* MIT Press, Cambridge, MA, USA, 1987.

[Sha89] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys (CSUR)*, 21(3):413–510, 1989.

[SJG94] Vijay Saraswat, Radha Jagadeesan, and Vineet Gupta. Foundations of timed concurrent constraint programming. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 71–80, 1994.

[Smo94] Gert Smolka. A foundation for higher-order concurrent constraint programming. In Jean-Pierre Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, Lecture Notes in Computer Science, vol. 845, pages 50–72. Springer-Verlag, 1994.

[SO08] Helmut Simonis and Barry O'Sullivan. Search strategies for rectangle packing. In *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming*, CP '08, pages 52–66, Berlin, Heidelberg, 2008. Springer-Verlag.

[SR89] Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245. ACM, 1989.

[SS94] Leon Sterling and Ehud Y. Shapiro. *The art of Prolog: advanced programming techniques*. Logic programming. MIT Press, Cambridge, Mass, 2nd edition, 1994.

[SS12] Joachim Schimpf and Kish Shen. ECLiPSe from LP to CLP. *Theory and Practice of Logic Programming*, 12(1):127–156, 2012.

[SSW09] Tom Schrijvers, Peter Stuckey, and Philip Wadler. Monadic constraint programming. *Journal of Functional Programming*, 19(06):663–697, November 2009.

[STF+10] Horst Samulowitz, Guido Tack, Julien Fischer, Mark Wallace, and Peter Stuckey. Towards a lightweight standard search language. In *The 9th international workshop on constraint modelling and reformulation (ModRef)*, 2010.

[STL14] Christian Schulte, Guido Tack, and Mikael Lagerkvist. *Modeling and Programming with Gecode*, 2014.

[STW+13] Tom Schrijvers, Guido Tack, Pieter Wuille, Horst Samulowitz, and Peter J. Stuckey. Search combinators. *Constraints*, 18(2):269–305, 2013.

[SVWSDK10] Jon Sneyers, Peter Van Weert, Tom Schrijvers, and Leslie De Koninck. As time goes by: Constraint handling rules. *Theory and practice of logic programming*, 10(1):1–47, 2010.

[SW12] Terrance Swift and David S. Warren. XSB: Extending Prolog with Tabled Logic Programming. *Theory and Practice of Logic Programming*, 12(1-2):157–187, January 2012.

[SWDD14] Tom Schrijvers, Nicolas Wu, Benoit Desouter, and Bart Demoen. Heuristics entwined with handlers combined: From functional specification to logic programming implementation. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*, PPDP '14, pages 259–270, New York, NY, USA, 2014. ACM.

[TA11] Charlotte Truchet and Gérard Assayag. *Constraint Programming in Music*. Wiley, 2011.

[Tac09] Guido Tack. *Constraint Propagation – Models, Techniques, Implementation*. PhD thesis, Saarland University, 2009.

[TAE17] Pierre Talbot, Carlos Agon, and Philippe Esling. Interactive computer-aided composition with constraints. In *Proceedings of the 2017 International Computer Music Conference, ICMC 2017,*

Shanghai, China, pages 265–270, Shanghai, China, 16–20 October 2017. Shanghai Conservatory of Music.

[Tar33] Alfred Tarski. Pojęcie prawdy w językach nauk dedukcyjnych. 1933.

[TC04] C. Truchet and P. Codognet. Musical constraint satisfaction problems solved with adaptive search. *Soft Computing*, 8(9), 2004.

[TDS03] Olivier Tardieu and Robert De Simone. Instantaneous termination in pure Esterel. In *International Static Analysis Symposium*, pages 91–108. Springer, 2003.

[Tec05] Esterel Technologies. *The Esterel v7 Reference Manual Version v7 30 initial IEEE standardization proposal.* Esterel Technologies, 2005.

[TFF12] Erich Christian Teppan, Gerhard Friedrich, and Andreas A. Falkner. QuickPup: A heuristic backtracking algorithm for the partner units configuration problem. In *IAAI*, 2012.

[TP17] Pierre Talbot and Clément Poncelet. Langage pour la vérification de modèles par contraintes. In *Treizièmes journées Francophones de Programmation par Contraintes, JFPC 2017, Montreuil sur Mer, France*, pages 201–211, 13–15 June 2017.

[Tri14] Markus Triska. *Correctness Considerations in CLP(FD) Systems.* PhD thesis, Vienna University of Technology, 2014.

[Tur37] Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 42:230–265, 1937.

[Ued85] Kazunori Ueda. Guarded horn clauses. *ICOT Technical Report, TR-103*, 1985.

[Ued90] K. Ueda. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, 33(6):494–500, June 1990.

[Ued09] Kazunori Ueda. LMNtal as a hierarchical logic programming language. *Theoretical Computer Science*, 410(46):4784–4800, 2009.

[Ued17] Kazunori Ueda. Logic/constraint programming and concurrency: The hard-won lessons of the fifth generation computer project. *Science of Computer Programming*, 2017.

[VH89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[VH91] P. Van Hentenryck. The CLP language CHIP: constraint solving and applications. In *Compcon Spring '91. Digest of Papers*, pages 382–387, Feb 1991.

[VH02] Pascal Van Van Hentenryck. Constraint and Integer Programming in OPL. *INFORMS Journal on Computing*, 14(4):345–372, November 2002.

[VHM99] Pascal Van Hentenryck and Laurent Michel. OPL script: Composing and controlling models. In *Compulog Net/ERCIM Workshop on Constraints*, pages 75–90. Springer, 1999.

[VHM05] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005.

[VHML05] Pascal Van Hentenryck, Laurent Michel, and Liyuan Liu. Contraint-based combinators for local search. *Constraints*, 10(4):363–384, 2005.

[VHMPR99] Pascal Van Hentenryck, Laurent Michel, Laurent Perron, and J.-C. Régin. Constraint programming in OPL. In *International Conference on Principles and Practice of Declarative Programming*, pages 98–116. Springer, 1999.

[VHPP00] Pascal Van Hentenryck, Laurent Perron, and Jean-François Puget. Search and strategies in OPL. *ACM Transactions on Computational Logic (TOCL)*, 1(2):285–320, 2000.

[VHSD91] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Constraint processing in cc(FD). Technical report, Brown University, 1991.

[VHSD98] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(FD). *The Journal of Logic Programming*, 37(1–3):139–164, 1998.

[VRBD+03] Peter Van Roy, Per Brand, Denys Duchier, Seif Haridi, Christian Schulte, and Martin Henz. Logic programming in the context of multiparadigm programming: The oz experience. *Theory Pract. Log. Program.*, 3(6):717–763, November 2003.

[WSTL12] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, January 2012.

[XHHLB08] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of artificial intelligence research*, 32:565–606, 2008.

[YA86] Rong Yang and Hideo Aiso. P-prolog: a parallel logic language based on exclusive relation. In *Third International Conference on Logic Programming*, pages 255–269. Springer, 1986.

[Zho06] Neng-Fa Zhou. Programming finite-domain constraint propagators in action rules. *Theory and Practice of Logic Programming*, 6(5):483–507, 2006.